Integration of Spatial Vector Data in Enterprise Relational Database Environments

Dissertation

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena
im Juli 2006

von
Dipl.-Inf. Knut Stolze
geboren am 8. Januar 1975 in Jena, Thüringen

Gutachter:

1. Prof. Dr. Klaus Küspert, Friedrich-Schiller-University Jena
2. Prof. Johann-Christoph Freytag, Ph.D., Humboldt-Universität zu Berlin
3. Dr. Albert Maier, IBM Deutschland Entwicklung GmbH, Böblingen

Tag der letzten Prüfung des Rigorosums: 7. November 2006

Tag der öffentlichen Verteidigung: 10. November 2006

To My Wife Annett

Abstract

Spatial and geometric data can be found in virtually every relational database. Such data is often not explicitly modeled but rather hidden in addresses, for example. Unless spatial data is treated as first class citizen in the relational database management systems (RDBMSs), spatial features cannot be leveraged and exploited by applications to their full extent. Several products provide the capabilities to store and query geometric data directly in relational databases. Those products include dedicated data types along with associated spatial functions and indexing mechanisms.

The SQL/MM spatial standard exists to ensure that the spatial interface provided by products are compatible. The standard covers basic spatial functionality only. Data types are defined and include a set of routines that are used to construct spatial values from external data formats, compare two geometries regarding their spatial relationship, generate new geometries, or to extract spatial properties. However, more advanced functionality is crucial for applications and their business logic is still lacking. Therefore, applications either disregard spatial information completely or noticeable effort is invested for the addition of customized spatial logic. This thesis is dedicated to closing the most significant gaps in the standard that we identified to further broaden the user base.

The first issue addressed in the thesis is the insufficient support of spatial data types in programming languages and the respective database connectivity mechanisms, most prominently Java and JDBC. We extend the JDBC interface to incorporate a spatial class hierarchy. Instances of these classes can be directly retrieved from the RDBMS or passed to it, avoiding the current clumsy and time-consuming stream-based interface.

Other major enhancements to the standard that are contributed by this thesis are graph and network related routines. We exploit the topological properties of geometries for functions that implement graph algorithms while treating graphs transparently as indexing structures for geometries only. Shortest path or traveling salesman problems can be solved directly on street networks. The results are in a relational fashion so that they can even be combined with data from other tables in the database.

The SQL/MM spatial standard is only concerned with two-dimensional data. This is insufficient. Many applications require at least a third dimension, for instance CAD/CAM. We develop and implement the concepts for the integration of 3D geometries into the standard. This work also comprises the necessary considerations for operations in true three-dimensional data space.

Enterprises operate diverse database systems and architectures. It is important to establish transparent access to spatial and non-spatial data alike in such environments. Federation and replication are two mechanisms to provide such access in distributed systems. We develop the techniques to integrate spatial data in federation and replication setups while relying on the available products only. This restriction is based on the fact that new and independent solutions and prototypes are usually not an option in more complex configurations.

Kurzfassung (German Abstract)

Räumliche und geometrische Daten existieren in praktisch jeder relationalen Datenbank. Diese Daten sind jedoch oft nicht explizit modelliert, sondern z. B. in Adressen versteckt. Aber nur die explizite Modellierung erlaubt die Nutzung der räumlichen Eigenschaften durch Applikationen. Verschiedene Produkte stellen Funktionalität zur direkten Verwaltung von räumlichen Daten in relationalen Datenbanken zur Verfügung. Es sind dedizierte Datentypen, sowie zugehörige Routinen und Indexmechanismen implementiert.

Zur Harmonisierung der Schnittstellen besagter Produkte wurde die SQL/MM Spatial Norm entwickelt. Allerdings definiert die Norm nur Grundfunktionalität. So existieren Datentypen und zugehörige Routinen für die Konvertierung zwischen Geometrien und zeichenbasierten externen Datenformaten, Routinen zum Vergleich zweier Geometrien gemäß ihrer räumlichen Beziehung, Routinen zum Erzeugen neuer Geometrien und Routinen zum Extrahieren von räumlichen Eigenschaften. Für Anwendungen sind aber höhere und komplexere Funktionen notwendig, damit räumlichen Daten nahtlos integriert werden können. Diese Arbeit widmet sich den wichtigsten dieser Lücken und macht Vorschläge, um sie baldigst in der Norm zu schließen.

Ein vordringliches Problem besteht bereits bei der direkten Einbindung der räumlichen Daten in die Schnittstellen zwischen Datenbanksystem und Anwendung, z. B. Java und JDBC. JDBC wird um eine Klassenhierarchie für räumlichen Daten erweitert. Instanzen dieser Klassen können direkt zwischen Anwendung und RDBMS ausgetauscht werden.

Weitere Modifikationen der SQL/MM Spatial Norm beziehen sich auf Graphen und Netzwerkoperationen. Die topologischen Eigenschaften von Geometrien werden für neue Funktionen ausgenutzt, wobei die Graphen transparent als Indexstrukturen Anwendung finden. Somit können kürzeste Pfade berechnet oder das Traveling-Salesman-Problem direkt auf den gespeicherten Straßendaten gelöst werden. Die Ergebnisse werden in relationaler Form aufgebaut, so dass sie direkt mit anderen Daten in der Datenbank verbunden werden können.

Die SQL/MM Spatial Norm bezieht sich ausschließlich auf zwei-dimensionale räumliche Daten. Für CAD/CAM Applikationen ist dies z. B. unzureichend, und zumindest eine weitere Dimension ist notwendig. Die Konzepte für die Integration von 3D Objekten in die Norm werden entwickelt und erprobt. Dies umfasst auch die Erweiterung aller bereits vorhandener Operationen für den drei-dimensionalen Raum.

Viele verschiedene Datenbanksysteme und Architekturen finden in kommerziellen Produktionsumgebungen Einsatz. Transparenter Zugriff auf räumliche und nicht-räumliche Daten ist in solchen Umgebungen essentiell. Föderation und Replikation sind zwei Mechanismen, um Daten zwischen den Komponenten verteilter Systeme auszutauschen. Diese Arbeit entwickelt die Techniken, um räumliche Daten in diese Prozesse zu integrieren, da sie bisher vernachlässigt wurden. Dabei wird gezielt auf die Möglichkeiten existierender Produkte gebaut, weil neue Lösungen und Prototypen üblicherweise nicht sicher in solch komplexen Konfiguration einzusetzen sind.

Contents

List of Figures

List of Tables

List of Listings

Preface

Structure of the Thesis The entire thesis consists of four parts. The first part is dedicated to the basics of spatial data in general and the SQL/MM spatial standard in particular. Chapter 1 gives a motivation on the deeper integration of spatial data in applications as well as an overview on the thesis. The concepts of spatial data in relational database systems are presented in Chapter 2.

The second part is specifically targeted at the gaps that are left open by the SQL/MM spatial standard. Each of the Chapters 3, 4, and 5 addresses one specific topic that has been neglected so far. The related literature, publications, and products are reviewed after an introduction on the goals of the chapter. Usually, the integration of the respective concepts into spatial extensions for relational database systems is at the same stage as the standard: there is none. Therefore, the concepts are driven forward to a point where they could be prototypically implemented. Based on that implementation, proposals are give how to close the gaps in the SQL/MM spatial standard.

The third part ventures beyond the scope of the SQL/MM spatial standard and investigates typical components that can be found in enterprise-scale deployments of relational database systems. Such environments are usually very heterogeneous and this aspect has been taken specifically into account for the handling of geometric data. Chapters 6 thru 8 describe the problems originating from the diversity of spatial RDBMSs and provide solutions to cope with that diversity. Each chapter explains the current situation in the spatial realm (which cannot be considered as satisfactory). The subsequently presented solutions tackle those problems.

The results and contributions of the thesis are summarized in the fourth and last part. Additionally, an outlook on possible future work is given.

Acknowledgments Many people supported and encouraged my work on this thesis during the past years. My thanks are extended to all of them, even if not everyone is mentioned here by name.

First and foremost, I would like to give my thanks to Prof. Dr. Klaus Küspert. He encouraged me to work in the area of spatial data and also sometimes applied the needed (but always friendly) pressure to proceed with the task in a timely manner. I am also grateful to Dr. Albert Maier and Prof. Dr. Johann-Christoph Freytag for their interest in the work. Both were immediately willing to act as secondary referees when asked. Both provided me early feedback that led to further improvements.

I appreciate the help of many students whose study and diploma theses, related the field of spatial database systems, I supervised jointly with Prof. Dr. Klaus Küspert. Raik Bittner, Christoph Salomon, Stev Witzel, Christian Zentgraf, and Andreas Bräu prepared excellent and valuable contributions. Also, the students Sebastian Ott, Martin Hoffmann, Martin Salzbrenner, Sebastian Niebius, Marcus Hetterle, Dennis Heimann, Matthias Walther, and Christian Schwartze have to be thanked. Without them, I would still not have finished the work.

Further thanks are extended to IBM and my managers there who gave me the time to pursue the research. Without the opportunity given to me by Nelson Mattos to work on and actively contribute to the DB2 Spatial Extender (Version 8), this thesis may never have been written in the first place. My co-workers Rafael Coss and David Adler were always a tremendous source for many ideas and solutions due to their deep knowledge of the spatial area and the requirements brought forward by customers who deployed the DB2 Spatial Extender in production environments and came up with new requirements that were finally incorporated in this thesis.

Last but not least, my sincerest thanks go to my family, especially my beloved wife Annett. She encouraged me at hard times with great patience. For that, she had to put up with all my quirks and also with my spending long hours on this work.

Conventions Various elements of SQL and programming languages are used in the text portion in the chapters. The following table lists how those elements are set, along with a short description for each.

Element	Description
`statement`	Many examples of SQL statements are used. If SQL keywords appear in the text portion, they are written in `monospace` font. The same applies to code fragments of C or Java programs.
name	Several listings contain SQL statements. The description in the text frequently refers to the elements in the listing, for example the names of SQL schemata, tables, columns of tables, data types, and attributes of structured types. Names of such objects are all set in sans-serif.
routine	The names of stored procedures, functions or methods on structured types are identified with *italics* font in the text.
NULL	NULL is a substitute in SQL to mark the absence of a value. It is not a value in itself. Due to its special meaning, it is emphasized in the text by SMALL CAPS.

group	Transform groups are SQL constructs to implicitly convert values of structured data types from or to another representation relying on the data types predefined by SQL only. The names of transform groups are written in *slanted* font.
class	Classes and interfaces are features of object-oriented programming languages like Java. The names of classes and interfaces are set in *slanted* font to distinguish them from the surrounding text.
method	Classes and interfaces in programming languages have methods associated with them. Such non-SQL methods are written in *slanted sans-serif*.
algorithm	*Italics* fonts are used to set the names of algorithm that are developed and described in the thesis.

Besides the before listed elements, the first reference to a figure, table, or listing is emphasized. *Italics* fonts are used for that. Thus, it is easier to find the description for a certain figure or table, which may not have been placed on the same page for type setting reasons.

Knut Stolze
Jena, Germany in July 2006

Part I

Basics

1 Introduction

Many application areas could take advantage of (or even require) storing, retrieval, and querying of geometric, geographic, and spatial data[1]. The data space for geographic data is the surface of the Earth, possibly as a two-dimensional abstraction. Spatial data is (nearly) always used together with other, non-spatial data. In geographic contexts, it models static objects in the real world like streets, buildings, political boundaries, etc. Moving objects like ships or cars with a changing location could also be described. In all practical situations, additional attributes are stored together with the location, for example, the name and direction of a street or the make, model, and owner of a car.

Geographic data plays an important role in many areas. For example, disaster control and emergency management need information about the exact location of a catastrophe or accident as well as the related infrastructure like cities, roads, railways, etc. Another major user group consists of military entities. Armies used spatial data for centuries already for their strategic and tactical planning. Espionage, reconnaissance, troop movements, and target selections during warfare are just a few scenarios where location information is crucial. Nearly all such systems employ relational database management systems (RDBMSs) to manage their data. Therefore, the handling of spatial data at the relational level is very important and has to be reflected in today's database technology.

Latitude and longitude values are oftentimes used for the coordinates of geographic data. Coordinates define the geometries with vectors. Many other, non-geographic scenarios exist. For example, genomic sequences in chromosomes need spatial information in a three-dimensional data space. Yet another typical usage of three-dimensional data can be found in computer-aided design (CAD) and computer-aided manufacturing (CAM) systems. Such systems do not operate on a geographic scale but rather in a much smaller one that is not related to the Earth. Applications of spatial data originating from CAD systems can easily be found. Motorcycle engines or the design of an integrated circuit board are just two examples.

The size and precision of the geometric data that is processed and managed by applications may cover a wide range. The required precision in a geographic scale is much smaller than the precision for engines. In one case, tolerances are measured in meters and centimeters and in the other these are fractions of millimeters. The complexity of a geometry, i. e. the number of points and the structure, is usually quite similar. A polygon describing a land parcel or a road can become quite complex – as are the paths of conductors on an integrated chip.

[1] Henceforth, the terms *geometry*, *spatial data*, and *spatial information* are used synonymously.

Geographic data, managed by geographic information systems (GISs), is by far the most important adopter of spatial data. GIS became more and more mainstream since the late 1980s [Bil99]. Their original focus laid on the storage and management of environmental information and utility data like water or power supply lines where the spatial data is the exact location of cables and pipes or the exact positioning of an oil well on the Earth's surface, for instance. The different types of spatial data and their associated non-spatial attributes were structured into so-called *layers*. Layers are comparable to entity types in the relational model since they describe similar objects, e. g. all highways. Usually, a single layer maps to a single table in an RDBMS.

The first GISs stored their spatial and non-spatial data in flat files. The storage format changed with the advent of RDBMSs [Sár01]. There have been early attempts to manage the data by RDBMSs due to the capabilities of the database systems to handle large amounts of spatial and non-spatial data using the relational model. Features like scalability, transactional properties and backup/recovery were exploited. With the introduction of object-relational (OR) features in the database world in the late 1990s, the geographic information systems were among the first serious adopters. The OR technology addresses the higher-than-normal flexibility and complexity requirements for the modeling of geometric objects like points or polygons [Mol98] while yielding better performance than the strict normalization of the geometric information. However, GISs only exploit very limited database functionality, basically treating the database management systems (DBMSs) as a slightly smarter file system with integrated indexing facilities. Major geographic information systems are, for example, ArcInfo [Mac99] or MapInfo [DLW02]. *Figure 1.1* shows a screen-shot of the graphical user interface of the open source system GRASS [GRA05]. The figure presents only a very small subset of GRASS, namely the visualization of layers of vector data.

Extending relational database systems with spatial data types, spatial functions operating on those types, spatial indexing, and spatial join mechanisms leads to *spatial database systems* [Güt94a]. Today, the use of spatial data and spatial database systems goes far beyond the original scope of GISs. A much wider audience and applications in virtually every domain are in the position to exploit spatial functionality. For example, location-based services (LBS) can be combined with the Global Positioning System (GPS) to provide advanced and up-to-date information to customers [SWCD97] of supermarkets or shipping companies. To that end, the company holds certain location information in its database and combines that with the position gathered from the GPS device. Cells of mobile phone networks can serve as another source for location information. The cells can be used to pinpoint the origin of an emergency call, for instance.

The SQL standardization committee recognized the need to harmonize the interface of spatial packages for relational database systems. The standard targeted at multimedia and application specific packages – SQL/MM – contains part 3 that is dedicated to spatial functionality. The development of the SQL/MM spatial standard [ISO03d] was initiated in the mid 1990s and the OpenGIS Simple Feature Specification for SQL

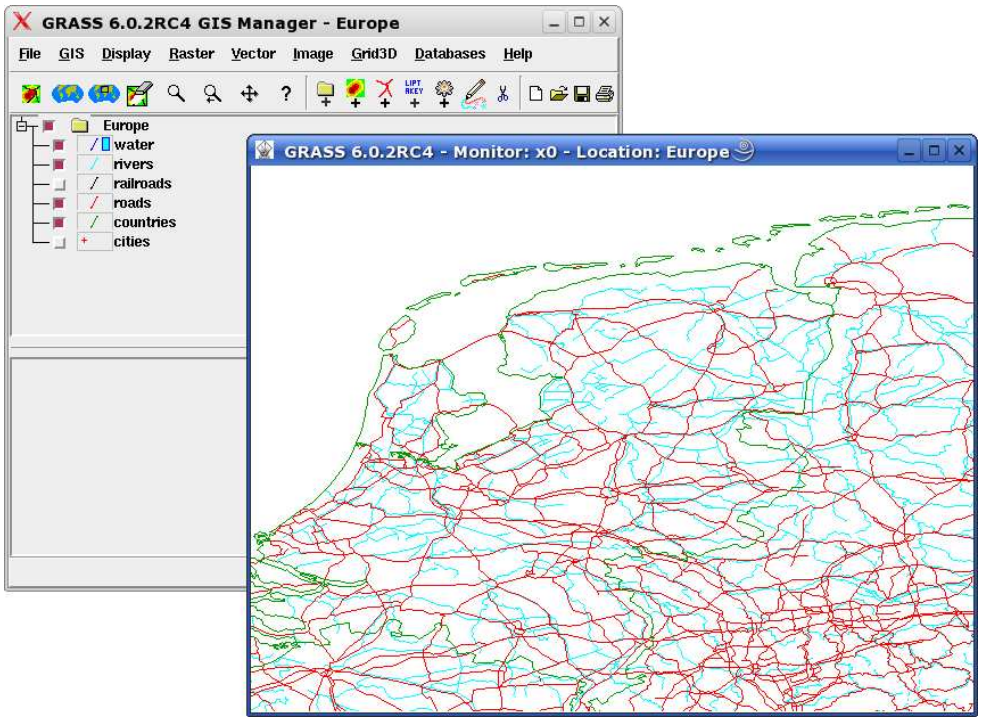


Figure 1.1: Screen-shot of the GRASS GIS

[OGC99] served as a solid foundation. Initially, the specification of the OpenGIS consortium (now renamed to the Open Geospatial Consortium) was taken as-is, then refined and built upon further. The standard describes the range of the facilities to store, manipulate, and retrieve spatial data in an RDBMS [Sto03b].

Although the SQL/MM spatial standard comes with a vast amount of functionality, it is not yet at a stage for every user and application to take full advantage of its definitions. Users without a GIS background are currently held back because the standard only covers basic building blocks. More advanced and higher-level functionality is not present. The primary exploiters remain GISs for the time being. The intention of the subsequent chapters is to address those functional gaps in the standard. We identify missing advanced functionality. Each chapter is dedicated to describe solutions to close those gaps. We measure the performance of the new techniques presented in this thesis, based on respective prototypical implementations.

The remainder of the thesis is structured as follows. Part I consists of this introductory chapter. Additionally, Chapter 2 presents the concepts of spatial data in depth. We describe the different types of geometries managed in relational database systems (for the two-dimensional data space). The SQL/MM spatial standard is introduced and discussed in all its major aspects. Several commercial and open source products available implement spatial packages (extensions) for relational database systems. We give an overview of the most important of these products and analyze product regarding its support of

and conformance to the SQL/MM spatial standard. None of the products addresses all needs for spatial computations in an appropriate way yet. That makes the area of spatial data management and processing in relational database systems an important research topic and we contribute solutions and results in the subsequent parts of this thesis.

We identify and close functional gaps in the SQL/MM spatial standards itself in Part II. Chapter 3 is concerned with the insufficient integration into language binding facilities like Java Database Connectivity (JDBC). Today, byte-stream-oriented data formats have to be used for the communication of spatial data between applications and the database system. Relieving applications of those transformations and establishing a direct communication of spatial objects between the systems is very much preferred. To that end, we explain how JDBC is extended for spatial data processing. We add a spatial class hierarchy to JDBC and objects of these classes can directly be retrieved from and send to the DBMS. Furthermore, the full range of spatial operations is made available to applications via the classes. So far, the IBM Informix Spatial DataBlade [IFX02b] is the only available product with a similar offering. However, our approach for the integration of spatial types into the language bindings exceeds its functionality.

Chapter 4 addresses the needs of business intelligence (BI) applications that require routing operations. The SQL/MM spatial standardization committee recognized the need for graph functionality and already adds a respective function for the upcoming version of the standard. We develop more sophisticated techniques to map spatial vector data to graph structures (and vice versa) so that graph algorithms can be applied transparently. For example, shortest paths or optimal round trips can be determined in street networks that are stored in the database as vector data. Applications are not burdened with the construction and maintenance of graphs. Instead, graphs are treated like an index mechanism on geometries. We implement an extension for relational database systems, the Spatial Graph Extender, to demonstrate the feasibility of the techniques. The only product that offers graph and network related functionality at the database level is the Oracle Spatial Topology and Network Data Models package [Ora05e]. However, the Spatial Graph Extender automates more of the processes for the graph construction and exploitation for graph operations.

Chapter 5 deals with the third major gap in the SQL/MM spatial standard, which is the lack of support for 3D geometries and operations in three-dimensional data space ($\mathbb{R}^3$). We extend the spatial type hierarchy with types for modeling 3D objects. All spatial operations are also carried to the higher-dimensional data space. Of course, that is in addition to the existing two-dimensional types and operations to retain backward compatibility. Thus, GISs can make use of the spatial functionality while CAD/CAM applications, which require the 3D functionality, are supported at the same time. The 3D extender, whose implementation is also presented in the chapter, is built on top of the DB2 Spatial Extender [IBM04d]. We show that the new functionality could be integrated successfully. That is in fact the first attempt of such an integration because none of the available products has ventured into this area yet.

Enterprise environments are quite complex and do not consist of a single (spatial) database system only. Therefore, Part III specifically considers such environments. At the beginning, we focus on the details of the implementations of the various spatial extensions in Chapter 6. The differences in the products show that access to spatial data managed in different database systems is not yet straight-forward. The independent management of spatial reference systems (SRSs) makes it necessary to map the SRSs between different database systems that are involved in a distributed system. We develop and present two mapping approaches. One of them takes advantage of the situation that most spatial extensions for RDBMSs accept SRSs in the so-called well-known text (WKT) format. The second approach is more general. It uses the schema developed by the European Petrol Survey Group (EPSG) as canonical format for SRSs.

Chapter 7 builds on Chapter 6 for federated systems. The spatial and non-spatial data residing in different databases is represented at the federated server in a global, consistent, and integrated schema. Such an integration is often needed in organizations with multiple groups where each group maintains its own database. The seamless access to geometry values from different data sources is established. Additionally, it is discussed how spatial predicates can be passed on to the respective data source where the spatial data actually resides. That means, filtering of data base on conditions involving spatial data is performed at the source. We implement a wrapper for the GRASS GIS [GRA05] to verify the techniques. By means of an example, we carry the results subsequently to the wrapper for DB2 data sources. We show that spatial data can be handled by the wrapper using new mechanisms to circumvent current restrictions in DB2 and the wrapper itself. Also, the push-down of spatial predicates could be achieved that way. An important finding is that no change to the source code of the (commercial) DB2 wrapper was necessary. Thus, the existing product can be used as-is and no new and potentially unstable wrapper is needed.

In Chapter 8 we deal with the integration of spatial data in replication configurations. Current products can handle spatial data only in homogeneous replication setups, i. e. when replicating from one database to another where both database systems are from the same vendor or product family [IFX05, Ora05b]. Such homogeneous environments are not very common and support for spatial data in heterogenous replication environments is required. Therefore, we develop the necessary strategies to handle spatial data with the existing and supported replication tools. Spatial data is converted to binary large objects (BLOBs), which are then replicated. An alternative approach builds on the fragmentation of an external representation for geometries so that the single fragments can be replicated like regular VARCHAR values. We demonstrate that both techniques are working successfully, while reducing the impact on the transactional processing occurring at the source system.

We conclude the work in the final Chapter 9 and summarize the results. We also discuss opportunities regarding the future development and improvement of the SQL/MM spatial standard and other areas that could potentially benefit from the integration and exploitation of spatial data.

2 Concerning Spatial Data

Two data models for spatial data exist since the rise of geographic information system (GIS): *raster data* and *vector data*. Raster data covers a certain area or region of the real world. The area is overlaid with a regular raster or grid. The raster partitions the area into raster cells with each cell storing just a single value. The amount of data may be influenced by the raster of the grid cells. The same concept is also known in graphics file formats like GIF or JPEG [BS95] where the cells are pixels containing color values.

Semantical connections between two points are not explicit in raster images. This information is handled completely different by spatial vector data. Points of geometries are explicitly connected by line segments or, more general, by curves. A single point in the database space, the coordinate system, specifies a location with respect to the origin of the coordinate system. A location is identified by its coordinate values for each dimensions of the data space. Therefore, a point is a *vector*. The mathematical concepts applicable to vectors are also applicable to spatial vector data.

We do not consider raster data any further in the subsequent chapters. It is a very broad topic that would exceed the scope of this thesis. It merits its own analysis regarding the integration in database environments. The concepts to manage image pyramids for varying resolutions, supporting multiple bands in images, and efficient management and retrieval of large images or portions thereof are just a few of those aspects.

Vector data, as it was originally implemented in GIS in the early 1980s, came through time with minor improvements only. Most notably, geometry values are encapsulated in an object-oriented fashion today, hiding the internal implementation. Additionally, a third dimension has been introduced to model altitude information.

We explain the concept of spatial vector data as it is referred to in the remaining chapters in depth in Section 2.1. The SQL/MM spatial standard builds heavily on object-relational (OR) features in SQL:2003 [ISO03i]; so the most prominent and relevant of those features are introduced in Section 2.2. An overview of the existing SQL/MM spatial standard itself is given in Section 2.3. The Open Geospatial Consortium (OGC) initiated the development of standards in the area of interoperable spatial data processing. Today, the focus of the OGC's activities lies on the development of the Geography Markup Language (GML). GML defines a series of XML schemata for the modeling of spatial data. The GML is not concerned with spatial functions and operations, so we discuss it briefly in Section 2.4. We close the current chapter in Section 2.5 with an overview of commercial and open source products that provide spatial functionality for relational database systems.

2.1 Spatial Data

Spatial data in relational database systems is stored as geometries. A geometry object is either a geometric primitive object or a combination of several geometric objects or primitives. Geometric primitives in two-dimensional space are points, curves, and surfaces. Lines and polygons are prominent subtypes for curves and surfaces, respectively.

In the field of GIS the term *geometry* is used to denote the geometric features that cartographers have used during the past centuries to map the world. That leads to the following *Definition 1*.

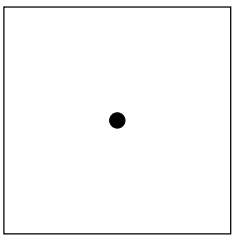
Definition 1 (Geographic geometry)
A geographic geometry is a point or aggregate of points representing a feature on the ground of the real world.

Today, a geometry is – usually – a model of a geographic feature because GIS are the primary users of spatial data. The model can be expressed in terms of the feature coordinates. The model conveys information; for example, the coordinates identify the position of the feature with respect to fixed points of reference like the equator together with the primary meridian or the center of the Earth.

2.1.1 Types of Geometric Primitives

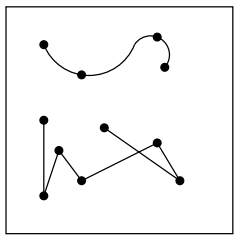
There are three different geometric primitives in two-dimensional space: points, curves and surfaces. The three types are intuitively understood but nevertheless they shall be introduced briefly.

Point



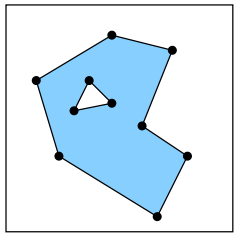
A point is the most simple geometry. A single coordinate value for each dimension identifies a point. For example, only an X and Y coordinate is present for points in $\mathbb{R}^2$. Points can be used in applications where the exact details of a feature are not available or not required depending on the maximum required resolution. If a company only requires the address of a customer and the location of his/her home with a precision of a few hundred meters, then a point is sufficient to represent the spatial location of the customer's home. Any detailed information about the ground plot of the house might be too excessive to handle in such cases (and unnecessary, too). Calculations with points are easier than operations on other geometric types due to the simple nature of points.

Curve Curves are the one-dimensional extension of points. Each curve connects at least two points. The points are also called *control points* or *inner points*. Streets or rivers are real-world objects often modeled with curves, for example. Such geographic objects actually cover an area of the Earth's surface. The curve representation is just an abstraction for reasonable low resolutions.



If the connecting curve between two points is a straight line, the curve is a linear string or linestring (also known as *polyline* [PS93]). Linestrings are very often sufficient and they also allow for easier spatial computations compared to arbitrary curves. General curves complicate the algorithms as, for example, the intersection between an arbitrary curve and another geometry cannot be calculated as easily. Also, algorithms in computational geometry are not yet available for all imaginable types of curves, much less the standardized external representations for them.

Surface A surface is a (geographic) feature that covers an extended area in a data space. A surface separates the data space into the interior and the exterior. The boundary of the surface is a closed curve. Applications for surfaces can be virtually any (projected) object on the Earth, assuming a sufficiently high resolution. Examples include forest areas, oceans or countries. If the boundary of a surface is described using linestrings, then the surface is a polygon. Computational geometry [PS93] analyzed extensively many different algorithms that operate on polygons.



2.1.2 Spatial Reference Systems

Each geometry is placed in a certain data space. Without the knowledge of the data space, the true meaning and representation of the geometry is unknown. The data space is commonly known as the coordinate system, or more specifically, the spatial reference system (SRS). Spatial reference systems became extremely important for GIS as the technology to measure the exact shape of the Earth improved since the late 19th century and, therefore, the location of objects on the Earth's surface could be determined more precisely. For geographic data, the shape of the Earth defines the spatial reference system as all locations are relative to the Earth's surface.

Three different kinds of coordinate systems are common in the GIS world. The coordinates of a geometry are either described using latitude and longitude, a true three-dimensional coordinate system with the Earth's center as the origin, or a two-dimensional coordinate system where the geometries are projected from the three-dimensional Earth on a flat, two-dimensional surface.

Geographic Coordinate Systems

A geographic coordinate system is a spatial reference system that employs a three-dimensional spheroid as surface on which the geometries are placed. Each point on the surface is described by its latitude and longitude. Thus, it is a spherical coordinate system.

The latitude ϕ specifies the location relative to the equator of the spheroid. The equator is the intersection of the plane defined by the major axis of spheroid with the spheroid surface. The latitude is measured in degrees in the interval $[-90, +90]$, where -90 and $+90$ degrees describe the south and north pole, respectively. Sometimes, cardinal directions are used, so that coordinates on the southern hemisphere are described as $n°South$ and, likewise, points on the northern hemisphere use $m°North$. The letters S and N may be used for abbreviations of *South* and *North*, respectively.

The longitude λ is measured relative to a chosen primary meridian. A meridian is a direct connection between the north and south pole on the spheroid surface. The primary meridian, also called the *prime meridian*, can be freely chosen for each spheroid, and it is an integral part of the definition of the geographic coordinate system. The values for the longitude are usually between $[-180, +180]$ degrees. Negative values for the longitude may also be specified using $x°West$ (or W in short for *West*), and positive values can be set as $y°East$ where *East* may be abbreviated with *E*. *Figure 2.1*, which was originally generated using the GMT toolkit [WS06], shows the location of the point $60°East, 48°North$ on a spherical map of the Earth's surface.

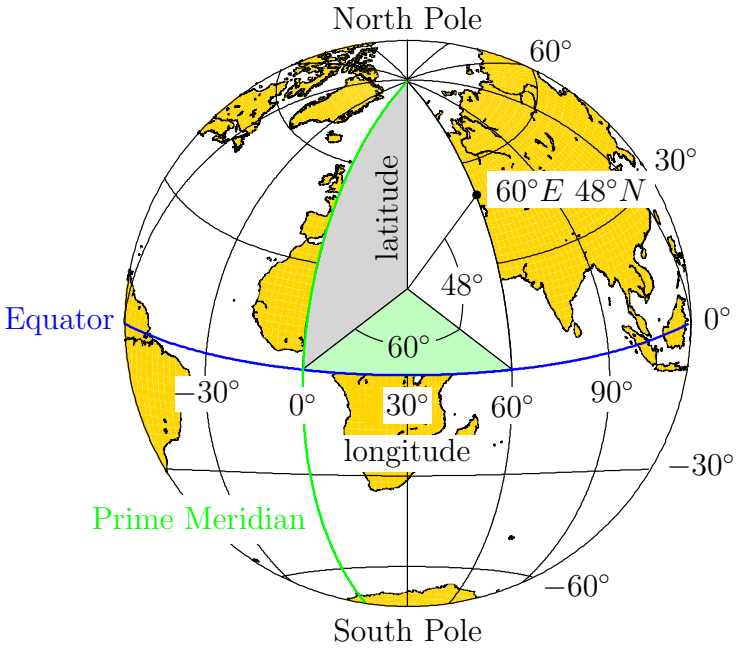


Figure 2.1: Example of a geographic coordinate system

Geocentric Coordinate Systems

Geocentric coordinate systems use also a spheroid, very much like geographic coordinate systems. However, the locations of points on the surface of the spheroid are given relative to the Earth's center using Cartesian X, Y, and Z coordinates and linear units for measurements. Therefore, a geocentric coordinate system is effectively a three-dimensional coordinate system. The restriction of the points to the surface of the spheroid allows an easy conversion between geographic and geocentric coordinate systems and vice versa. An example of a geocentric coordinate system and the placement of the point $(4100, 2367, 4263)$ on a sphere with radius $6371km$ (the average radius of the Earth) is illustrated in *Figure 2.2*. This point is exactly the same as in Figure 2.1.

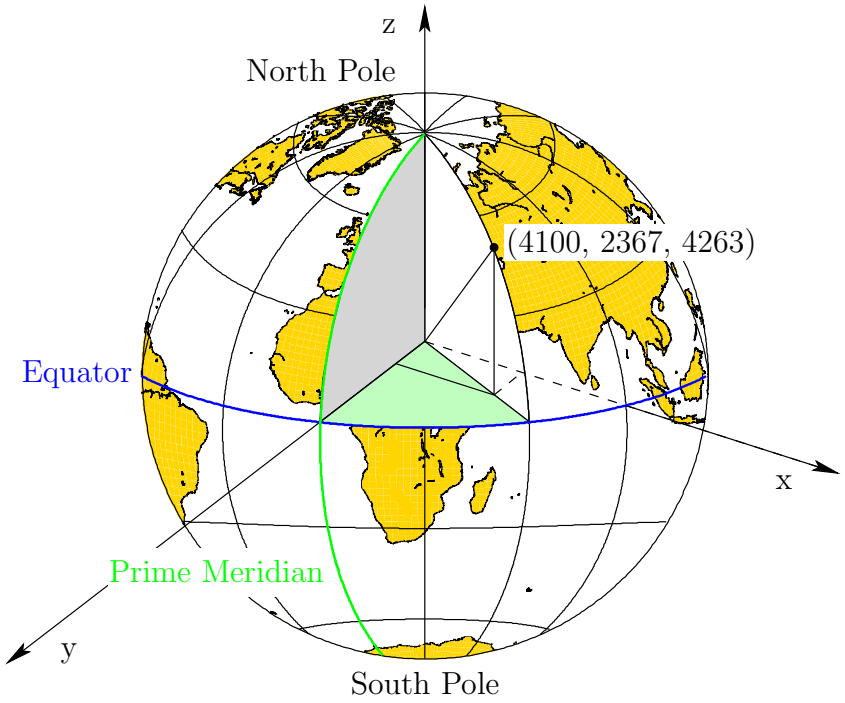


Figure 2.2: Example of a geocentric coordinate system

Projected Coordinate Systems

Projected coordinate systems were invented to be able to draw the Earth's round surface and the features on that surface on a flat, two-dimensional sheet of paper. A projection is an algorithm that describes how exactly the coordinates of a point on the Earth's surface needs to be converted to a point in the two-dimensional system. Many different algorithms and mathematical functions for projections exist, for example conical, cylindrical, or Mercator-based [KK00]. *Figure 2.3* shows the Mercator projection of the Earth

and its continents. The latitudes are in the interval $[-78°, 78°]$ because the projection stretches to infinity while getting closer the poles at $\pm 90°$.

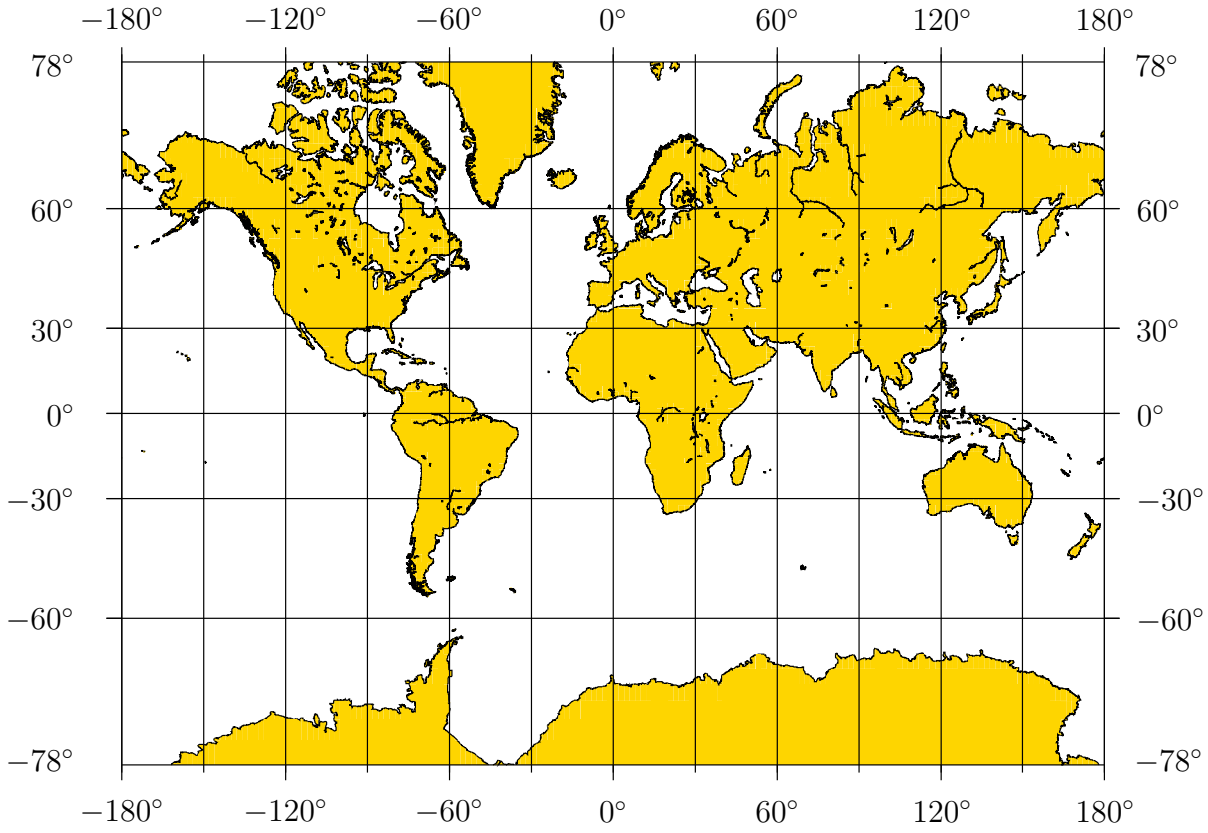


Figure 2.3: Example of a projected coordinate system

Each projection has specific properties and comes with different distortions compared to the geographic or geocentric coordinate system. Some projections preserve the area of geometries, others the direction or the perspective. None of the projections can ensure that distances measured in the projected coordinate system matches with the real world distances in the geographic coordinate system. A very good example is obvious from Figure 2.3. The north and south pole are just single points in the real world, but in the projection these two points are only reached in infinity and both would expand over the whole width of the figure.

Other Coordinate Systems

Geometries are not restricted to represent geographic features. The SRS will not be related to a geographic coordinate system in non-geographic applications. A geometry can identify, for example, a feature in an image where no relation to the Earth can be

established. As an example, the photography of a molecule contains inherent position information for the single atoms, and those positions could be spatially evaluated [Ign01]. Another example are locations inside a grocery store. Although a relation to the Earth could be computed based on the latitudes and longitudes, a preferred representation for an application might only refer to the locations with respect to a fixed point in the store, e. g. the south-east corner.

2.1.3 Properties of Geometries

Each geometry has – regardless of its spatial reference system – various properties. The more important properties shall be explained briefly as they are relevant in subsequent sections and chapters.

Geometry type Each geometry has an inherent type that identifies it as a point, line, polygon or a collection thereof. The type information is usually used when geometries are converted between different representations or to determine if a type-specific algorithm must be applied. An example for such type information are the numeric identifiers in the well-known binary representation [ISO03d].

Dimension Depending on the type of a geometry, the dimension is implicitly given. A geometry can have the following dimensions:

0 The geometry is a point, i. e. it is not linear and it covers no area or volume.

1 The geometry is linear but does not cover an area. Curves and linestrings have this dimensionality.

2 Surfaces have a dimension of two and they occupy an area.

Additionally, the dimensionality of -1 is sometimes used to classify an empty geometry, which might be the result of the intersection of two disjoint polygons.

Interior, exterior and boundary A geometry describes a set of points in the respective data space. Those points are the *interior* of the geometry. The remaining portion of the whole data space, i. e. the part that is not occupied by the geometry, is called the *exterior*. Interior and exterior are separated by the geometry boundary.

Coordinates Coordinates are the means to define the boundary of a geometry. They consist of a finite set of points (in $\mathbb{R}^2$ given by their X and Y coordinates) and implicit or explicit conventions how those points are to be connected.

Closed Curves can be closed or not. A curve is closed if its first and last point are identical. Naturally, the boundaries of polygons must be closed and not self-intersecting curves. Such curves are also known as *rings*.

Simple Some geometry types allow to distinguish between simple and non-simple geometries. For example, linestrings are simple if they don't intersect themselves. Simple polygons are polygons without holes in their interior, i. e. the boundary consists of a single, closed curve.

Minimum bounding box The minimum bounding box (MBB) is the minimum axis-parallel box that fully contains the geometry. The MBB is a minimum bounding rectangle (MBR) in two-dimensional space. Depending on the geometry itself, the MBB might collapse to a degenerated box. A single point only has a degenerated MBB, the point itself. The second exception are axis-parallel linestrings where the MBB is just that linestring. Computing the MBB in geographic or geocentric coordinate systems is not trivial as the geodetic line that connects two points on the surface of the spheroid is not necessarily parallel to the meridians or the equator – a minimum bounding circle (MBC) is often used instead.

The above list does not comprise the full set of properties. Further spatial properties might be relevant and used in specific applications.

2.2 Object-Relational Features in SQL:2003

The integration of spatial data into relational database systems originally began with the normalization of the geometry information [OGC99]. Point data was stored in separate tables, the so-called geometry tables. Each row in a geometry table contained a certain limited number of points. Thus, the definition for each geometry could comprise a set of rows. GIS products additionally introduced a second table for indexing purposes. This table stored information of a spatial index, for example an R-Tree [Gut84] or a Quad-Tree [Sam84].

The normalized storage model is a very inefficient and low-performing approach to manage spatial data because the access to the geometry data requires additional join operations and table scans in SQL statements. Therefore, large objects (LOBs) are a second approach described in [OGC99]. The geometry information is still stored in a separate table, but all points of a geometry are combined into a single binary large object (BLOB), encoded in well-known binary (WKB) format, for example. We describe the WKB format in more detail in Section 2.3.3.

The separation of the geometry information into secondary tables could not be considered as an ideal solution as it requires the access to multiple tables at query time. Options to address this issue became available with the rise of the object-relational (OR) features in some of today's database management systems [Luf05]. The first step to standardize those features was made as part of SQL:1999 [ISO99]. The following and now current SQL:2003 standard [ISO03i] did refine and further extend those features. The basic OR features described in the SQL standard are the following:

User-defined data types SQL defines a set of predefined data types, including INTEGER, CHARACTER VARYING, DOUBLE PRECISION, or BLOB. Such data types, although very generic, are not adequate for all applications. Especially spatial data has its own requirements on the type system available in a relational database system. To address those needs, SQL allows the creation of user-defined types to extend the predefined type system.

User-defined types in SQL:2003 can be simple *distinct types*, *structured types*, or *collection types*. Structured types are very similar to classes in object-oriented programming languages like Java [GJSB05]. A structured type consists of a set of attributes. The object-oriented paradigms of encapsulation, inheritance and dynamic typing are available as well. Structured types can be nested (but not recursively), and values that are instances of structured types are treated like any other scalar value in the database system.

Typed tables and typed views Typed tables (and also typed views) are defined by means of structured types. The names and data types for the columns of the typed table are derived from the attributes of the structured type. If the structured type on which the type table is based has an attribute A of type B, then the typed table has a column A of type B. Additionally, each typed table carries a reference column – the object identifier (OID) column – to uniquely identify the rows in the table.

The tuples in a typed table are effectively instances of the structured type. Thus, the inheritance rules of structured types are applicable as well, leading to table hierarchies or view hierarchies. However, storing values of structured types in a typed table looses the encapsulation as the internal structure of the type is made public as columns of the table.

User-defined routines The initial steps to extend a relational database system were user-defined routines. Routines allow the definition of operations on user-defined and predefined data types. Thus, application logic can be moved inside the database system, closer to the stored data itself.

Functions can be used in any SQL statement like any other expression. Some database systems do support the BOOLEAN data type as return type for function and that allows functions to be used directly as predicates. The logic of a function can be implemented using set-oriented or procedural SQL or a supported external programming language like C [KR88]. A special case for user-defined functions are implicit or explicit *cast functions*.

Methods are very similar to functions, but they are always tied to a structured data type. Each method has an implicit subject parameter `SELF` that refers to the instance of the structured types on which the method is invoked.

Functions and methods are limited with respect to the constructs that can be used in the routine body, i. e. no data modifications are allowed. *Procedures* lift that restriction, but they cannot be used directly in queries or SQL statements for data modification. Procedures must be invoked with the `CALL` statement. They can be called from triggers, other procedures, or directly from an application.

Language bindings Any data stored in a database system has to be imported, loaded or inserted into the tables. Likewise, the data needs usually to be extracted to be processed in an application. Transferring values of structured types between applications and the database system is more complicated than for the predefined data types. The structures used in the application might differ from the structured type. Furthermore, language bindings like ODBC or embedded SQL are tailored to transfer simple scalar values.

The communication of structured types in accomplished with standardized SQL through the use of so-called transform groups. A transform group combines the functions to create a value of a structured type from a value of a predefined data type and a function to convert a structured type to such a value. These `TO SQL` and `FROM SQL` functions use a serialization of the structured data.

Access to external data sources The shared and concurrent access to data stored in different source systems becomes more important in the industry. The source systems might be relational or not. The SQL/MED standard [ISO03l] specifies the infrastructure to manage the access to files external to a database system using *datalinks* and to represent external data as relational tables.

[TS06] considers large objects (LOBs) as OR feature. However, the standardized CLOB is a predefined data type to store character strings as is the type CHARACTER VARYING[1]. The differences are just other restrictions on the maximum length and that CLOB values can be accessed from an application with locators. Therefore, a distinction between both types with respect to OR is a rather artificial one. BLOB values are a binary variation of CLOB values to handle binary strings that include non-printable characters or other binary data. Similarly, assertions, constraints, and triggers as mechanisms to enforce integrity or business rules are sometimes considered as object-relational features. Those features are orthogonal to the above listed OR functionality.

The OR features relevant for the SQL/MM spatial standard comprise user-defined structured types, functions and methods that operate on those data types, and language bindings. Structured types are used to model the spatial type hierarchy and to encapsulate the storage of the definition of geometries. Transform functions are an integral part of the language bindings. The details of the SQL/MM spatial standard are introduced in the following section.

[1] SQL:2003 treats the data type names CHARACTER VARYING and VARCHAR as synonyms.

An in-depth description of OR features, in particular the mechanisms for collection handling, is given in [Luf05]. Lufter analyzes the current state of the support for those features in the SQL standard as well as the available database management systems.

2.3 The SQL/MM Spatial Standard

ISO/IEC 13249 SQL/MM is the effort to standardize extensions for multimedia and application-specific packages in SQL. It effectively belongs to the SQL standard as defined in [ISO03i], which plays an important role in the database industry. SQL is extended to manage data like texts, images, spatial data, or to perform data mining. The SQL/MM standard is grouped into several parts.

Part 1 is the framework for all the subsequent parts and specifies the definitional mechanisms and conventions used in the other parts as well the common requirements that an implementation[2] has to adhere to if it wants to support any of the extensions defined in the standard. Part 2 is the full-text standard, which is concerned with the mechanisms to provide extended text search capabilities, above and beyond the operators provided natively by SQL, e. g. the `LIKE` predicate. Part 5 defines the functionality to manage still images, and part 6 is addressed with data mining. The withdrawn part 4 was intended for general purpose facilities.

ISO/IEC 13249-3 SQL/MM Part 3: Spatial [ISO03d] is the International Standard (IS) that defines how to store, retrieve and process spatial data using SQL. It defines how spatial data is to be represented as values in a relational database, and which functions are available to convert, compare, and process this data in various ways. The first version of the standard was published in 1999. In the years since then, several enhancements were added to the document and the second version is now available as International Standard (IS). The process for developing the third version is on the way. The current working draft of the standard already contains some additions, for example to define points in geometries with three or even four dimensions, i. e. the addition of Z and M coordinates that has been neglected so far. Another improvement is the new routine *ST_ShortestPath*, which provides routing operations on linestring data. Both additions are highly relevant to the Chapters 4 and 5. We explain that the support for 3D and graph operations in the current working draft still falls short. Therefore, both concepts are extended beyond the scope of the working draft until a point is reached where the functionality is actually usable and beneficial.

The SQL/MM spatial standard is divided into clauses. The clauses 5 thru 9 describe the geometry types and the methods provided for each type. Like any other part of SQL/MM, the spatial standard contains an Information Schema. The Information

[2] The SQL standard uses the term *implementation* to refer to a program that implements the interfaces defined by the standard. Commonly, an implementation is a relational database system.

Schema, based on a Definition Schema, is defined in clause 14. The remaining clauses explain the underlying spatial concepts, the angles and direction handling, and the status codes and conformance rules for products that implement the standard.

The following sections are organized as follows. We give a brief history of the development of the SQL/MM spatial standard in Section 2.3.1. An introduction to the data types and how they are organized along with their methods follows in Sections 2.3.2 and 2.3.3. That lays the foundation for all subsequent chapters. The data types for angles and directions as well as for spatial reference systems are summarized in Section 2.3.4. The status codes and conformance rules are not explained, but we present the spatial information schema in Section 2.3.5. It is shown which information can be retrieved from its views. Two short examples illustrate the use of the spatial functionality in a relational database in Section 2.3.6. This rounds up the overview of the SQL/MM spatial standard.

Each of the sections summarizes a certain aspect in the standard. As it is the case with any standard related to SQL, there is always room for improvements. We discuss the standardized functionality critically in Section 2.3.7 and propose how the issues could be addressed. Most notably, the spatial type hierarchy could be changed to better model spatial data based on usability instead of implementation aspects. Therefore, Section 2.3.8 introduces a remodeled type hierarchy and explains its benefits. Besides some sub-optimal design decisions for the concepts, there are several areas that are not incorporated in the standard at all. Section 2.3.9 gives a brief introduction to some of the major areas, whereas the following chapters of this thesis are concerned with a more detailed explanation as well as proof-of-concept implementations.

2.3.1 History

The roots of the SQL/MM spatial standard are directly apparent in its type hierarchy. The standard was originally derived from the OpenGIS Simple Features Specification for SQL (SFS) [OGC99], which was published in 1999 as Version 1.1 by the OGC, formerly known as the OpenGIS Consortium. The Simple Feature Specification defines a *Geometry Model* that was further refined in [ISO04a]. The geometry model consists of a class hierarchy and a description for the methods applicable to each class. The hierarchy is shown in *Figure 2.4* using the Unified Modeling Language (UML). Inheritance relationships use a triangle along their lines where the upper point of the triangle is directed towards the superclass. The single association between the classes *Geometry* and *SpatialReferenceSystem* is represented by a simple line. The remaining relationships are aggregations and a diamond is attached to the aggregating class. The numbers right next to the lines indicate the number of elements that are aggregated, e. g. at least two points go into a linestring. In all other cases, at least one object is to be aggregated. Classes, whose names are set in italics, are abstract.

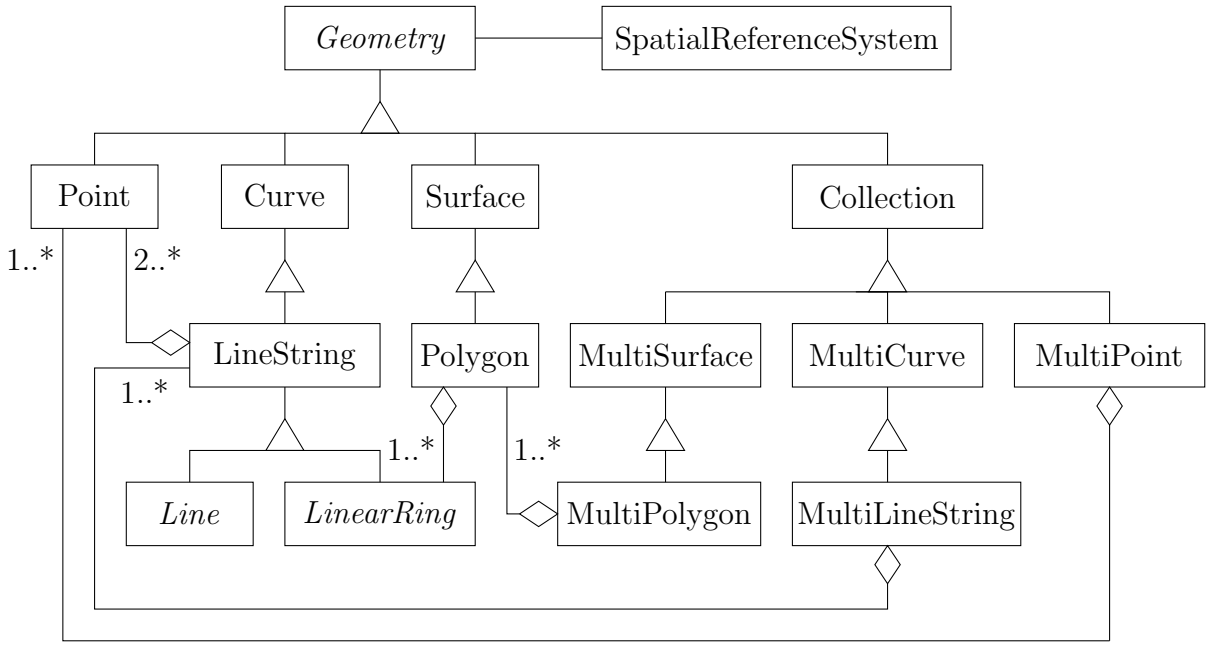


Figure 2.4: Geometry class hierarchy in the OpenGIS Simple Feature Specification

The geometry model of Figure 2.4 is an abstract model. It is used to define the classes for the geometric primitives *Point*, *Curve*, and *Surface* and collections thereof. The relationships and inheritance rules between the various classes are apparent in the class diagram in Figure 2.4. The inheritance rules are important for the methods defined on the classes and subclasses. For example, the method *Area* is defined for the class *Surface* and, therefore, is available for all instances of *Surface*, *Polygon*, and further subclasses, whereas the method *ExteriorRing* is only defined on the subclass *Polygon* and, thus, cannot be used for arbitrary instances of the class *Surface* or other geometries.

The OGC geometry class hierarchy implements the *composite* design pattern [GHJV95] by adding the class *Collection*. Without the subclasses under *Collection* and an additional aggregation of *Geometry*, the pattern would fit exactly. A simplification of the hierarchy could have been achieved by omitting all subclasses of Collection without any loss of functionality. The methods that are defined on the subclasses of *Collection* have all the same simple logic. The method is invoked for each element (also called *part* or *item*) of the collection and the results are combined. For example, the area of a multi-polygon geometry is calculated by summing the areas of the single polygons in the collection. Another drawback of the SFS's way of modeling the classes is that a future extension of the geometry model with new geometric primitives (for example to support solids as Chapter 5 demonstrates) requires the handling of the new classes in two different places. First, a class for the new geometric primitive itself has to be added under *Geometry*, and then a corresponding class is needed for collections of such geometries under *Collection*.

2.3.2 Spatial Type Hierarchy

The OGC geometry class hierarchy laid the foundation for the corresponding SQL type hierarchy that is defined in the SQL/MM spatial standard. *Figure 2.5* shows the standardized SQL type hierarchy. The types with the shaded background are not instantiable. It is implementation-defined whether ST_MultiCurve and ST_MultiSurface are instantiable or not, even though they are shown as not-instantiable in the figure. All types are used to represent geometric features in the two-dimensional space ($\mathbb{R}^2$). The major differences between the SQL/MM type hierarchy and the OGC geometry class hierarchy are the omission of the derived types Line and LinearRing, and the addition of a series of curve-related types. Lines and linear rings are to be represented using values of type ST_LineString, which covers both cases. The new types extend the OGC geometry class hierarchy with circular arcs as curves and surfaces that have circular arcs as their boundary. There are no commercial products available that implement those as explicit types, yet. Furthermore, the aggregations that reflect which types are used by other types are not shown. For example, it is not obvious from the SQL type hierarchy in the figure that the type ST_MultiPoint is a collection of ST_Point values only.

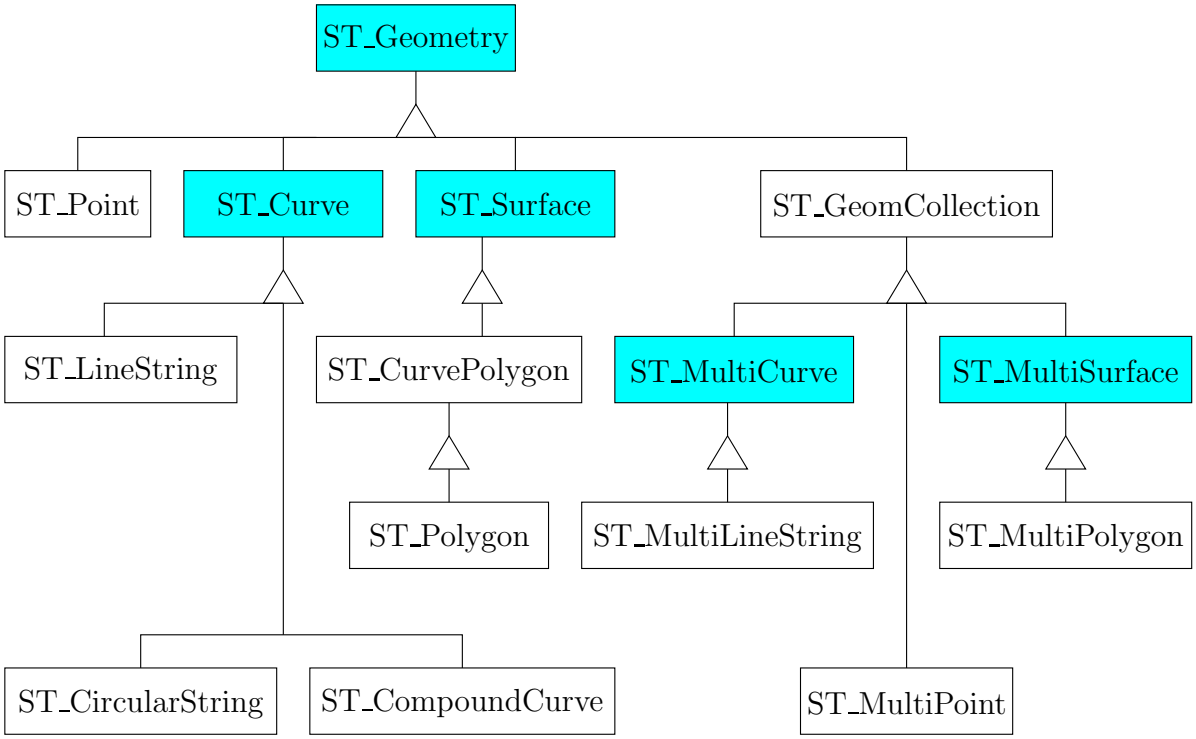


Figure 2.5: SQL/MM spatial type hierarchy

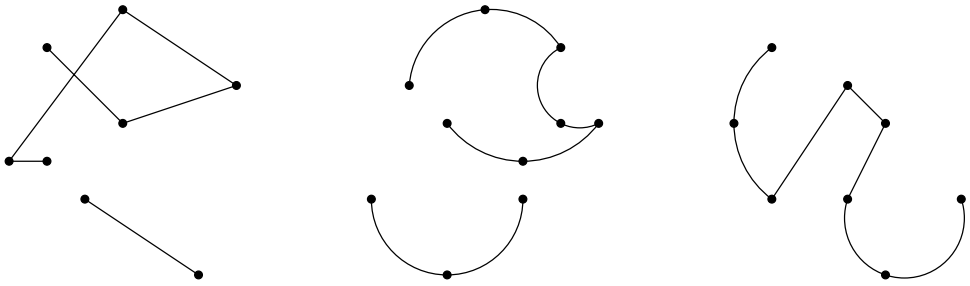
Another difference can be found in the naming conventions. The SQL/MM spatial standard uses consistently the prefix `ST_` for all tables, views, types, methods, and function

names. The prefix stood originally for *Spatial* and *Temporal*. It was intended in the early stages of the standard development to define a combination of temporal and spatial extension. A reason for that was that spatial information is very often tied with temporal data when objects are tracked in space *and* time [SWCD97, TJS97, GBE+00]. During the development of the SQL/MM spatial standard, it was decided that temporal has a broader scope beyond the spatial application and should be a part of the SQL standard as SQL/Temporal [ISO01], primarily based on [Sno95]. The contributors to SQL/MM did not want to move forward with a Spatio-temporal support until SQL/Temporal developed. However, SQL/Temporal was not developed any further and, like SQL/MM Part 4 [ISO98], it was subsequently withdrawn completely. In the meantime, the SQL/MM spatial standard focused on keeping it aligned with the OGC specification and the standards developed by the technical committee ISO/TC 211, for example [ISO03a, ISO03b]. The prefix `ST_` for the spatial tables, types, and methods was not changed during the reorganization, however. Today, one might want to interpret it as *Spatial Type*.

ST_Point values are 0-dimensional geometries and represent only a single location. Each point consists of an X and a Y coordinate to identify the location in the respective spatial reference system. Points can be used to model small real-world objects like lamp posts or wells. ST_MultiPoint values stand for a collection of single points. The points in a multi-point do not necessarily have to be distinct points. Thus, a multi-point supports multi-sets like SQL does in general.

Curves are 1-dimensional geometries. The standard distinguishes between ST_LineString, ST_CircularString, and ST_CompoundCurve. An ST_LineString is defined by a sequence of points – (X, Y) pairs – that are the reference points (or inner points) of the line string. Linear interpolation between the reference points defines the resulting linestring. That means, two consecutive points create a line segment in the linear string. Circular instead of linear interpolation is used for ST_CircularString values. Each circular arc segment consists of three points. The first point defines the start point of the arc, the second is any point on the arc, other than the start or end point, and the third point is the end point of the arc. If there is more than one arc in the circular string, the end point of one arc acts as the start point of the next arc. A combination of linear and circular strings can be modeled using the ST_CompoundCurve type. Line segments and circular segments can be concatenated into a single curve. *Figure 2.6* illustrates some examples of the three different types of curves. ST_MultiCurve values represent a multi-set of ST_Curve, and ST_MultiLineString a multi-set of ST_LineString values. Note that there are no types ST_MultiCircularString and ST_MultiCompoundString in the hierarchy. The reasons for that inconsistency are not clear anymore.

Surfaces, as 2-dimensional geometries, are defined in the same way as curves by using a sequence of points. The boundary of each surface is a curve. If the surface has holes in its interior, a set of curves is needed where the first curve describes the exterior, outer boundary. All further curves define holes. Each curve in the boundary is a ring, which



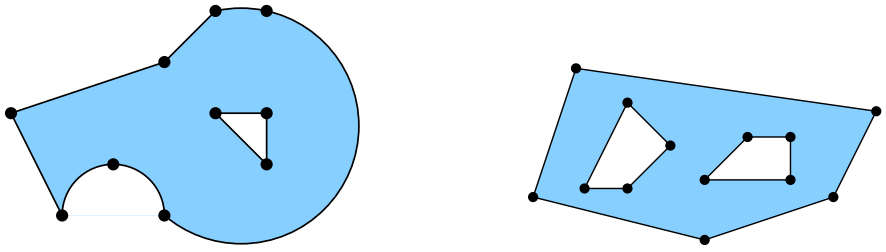
(a) Linear strings (b) Circular strings (c) Compounds

Figure 2.6: Examples of curves

means it is closed (the start point is identical with the end point of the curve) and it does not intersect itself. The start and end points of a curve form the boundary of a non-closed curve. A closed curve has an empty boundary, per definition. Thus, a curve c is a ring if and only if the following condition holds true:

$$(p_i, p_j \in Interior(c) : i \neq j \rightarrow p_i \neq p_j) \wedge Boundary(c) = \emptyset$$

The type ST_CurvePolygon stands for a generalized surface where the boundary of the polygon can be any supported curve, i. e. linear strings, circular strings, or compound curves. The subtype ST_Polygon restricts the conditions for the rings of the boundary to linear strings only. The types ST_MultiSurface and ST_MultiPolygon are used to model sets of curve polygons or sets of polygons with linear boundaries. ST_MultiSurface constrains its values to contain only disjoint surfaces. The last condition implies that a polygon in $\mathbb{R}^2$ has to be represented by a single polygon if it describes a single connected area. Examples of a curve polygon and a polygon with linear strings as its boundary are illustrated in *Figure 2.7*. The boundaries are shown as black lines and the interior of the polygons is shaded.



(a) Curve polygon (b) Polygon

Figure 2.7: Examples of polygons

2.3.3 Methods on the Spatial Types

There are a total of 142 spatial functions and methods defined in the SQL/MM spatial standard. The majority of them can be grouped into one of the following four categories:

- convert between geometries and external data formats,
- retrieve properties or measures from a geometry,
- compare two geometries with respect to their spatial relationship, and
- generate new geometries from others.

Examples and descriptions for each of the categories are given in this section. Sometimes, it is not trivial to assign a spatial method to a single category only. For instance, the method *ST_StartPoint*, which returns the first point of a linestring, retrieves a property of the linestring, i. e. the first point, but it also generates a new geometry, i. e. an ST_Point value.

Convert to and from External Data Formats

The SQL/MM spatial standard defines three external data formats that can be used to represent geometries in an implementation-independent fashion:

- well-known text (WKT) representation,
- well-known binary (WKB) representation, and
- Geography Markup Language (GML).

Each geometry type implements constructor methods that generate a new value from the given WKT or WKB and the, optionally, provided numeric spatial reference system identifier. All instantiable types have such constructor methods. There are no constructor methods that cope with the GML representation. Functions like *ST_LineFromGML* or *ST_MPointFromGML* are used instead.

For backward compatibility, the standard also defines functions like *ST_PointFromText* or *ST_GeometryFromWKB* with exactly the same purpose as the constructor methods. Those functions were inherited from and remain for compatibility with the OpenGIS Simple Features Specification for SQL [OGC99]. The constructor methods were introduced later in the process of the standard development. During the work on the second version, it was decided to align part 3 of SQL/MM with other parts and also to improve the overall usability by defining said constructors.

The three methods *ST_AsText*, *ST_AsBinary*, and *ST_AsGML* are provided for the conversion of a geometry to the respective external data format. The result is a character large object (CLOB) or BLOB value that contains the proper encoding of the geometry.

Retrieve Properties

All geometries have certain properties as we already introduced in Section 2.1.3. A property is, for example, the dimensionality or the information if a geometry is empty. Each subtype in the type hierarchy adds further, more specific properties. For example, the area of a polygon or whether a curve is simple, i. e. if it is not self-intersecting. A set of methods was defined to query those properties. Due to the high number of available methods, only a small set of examples is given here. All others can be found in [ISO03d].

ST_Boundary returns the length of the boundary of a surface or multi-surface geometry.

ST_IsValid tests whether a geometry is valid, i. e. correctly defined; an invalid geometry could be a non-closed polygon.

ST_IsEmpty tests whether a geometry is empty.

ST_X returns the X coordinate of a point.

ST_IsRing tests whether a curve is a ring, i. e. the curve is closed and simple.

ST_Length returns the length for a linestring or multi-linestring.

ST_Perimeter returns the length of the boundary of a polygon or multi-polygon, which is semantically the same as invoking `ST_Boundary().ST_Length()` on a polygon or multi-polygon.

ST_NumGeometries returns the number of geometries in a collection.

Compare Two Geometries

A very interesting part in spatial queries stems from the ability to spatially compare geometries. Questions like "which buildings are in a flood zone" or "where are intersections of rail roads and streets" can only be answered if the geometries representing the buildings, flood zones, rail roads, and streets are compared with each other. Effectively, such comparison operations are the means to formulate spatial joins [Ore86, Gün93].

Spatial data comes with a special property that was first summarized as Tobler's first law in 1970 [Tob70]:

> *Everything is related to everything else, but near things are more related than distant things.*

This law implies that various spatial predicates are needed to identify when a relation between two objects is deemed to be relevant. Therefore, the standard defines the following binary methods to compare geometries in various ways.

ST_Equals tests the spatial equality of two geometries.

ST_Disjoint tests whether two geometries do not intersect at all.

ST_Intersects, ST_Crosses, and ST_Overlaps test whether the interiors of the two given geometries intersect. The difference between the three methods lies in the allowed combination of input geometries, e. g. a linestring may not overlap a polygon but it can intersect or cross it.

ST_Touches tests whether two geometries touch at their boundaries but do not intersect in their interiors.

ST_Within and ST_Contains tests whether the first geometry is fully within the other, or if the first geometry fully contains the second.

ST_Relate tests the spatial relationship of two geometries based on Egenhofer's Nine Intersection Model [EH90]. The relation of the interior, boundary, and exterior of both geometries are pairwise compared and tested for the dimensionality of the respective intersection.

All of the above methods return an INTEGER value, which is 1 (one) if the spatial relation does exist and 0 (zero) otherwise. Ideally, the type BOOLEAN would have been chosen as return type, but that data type is not mandatory for database systems that conform to [ISO03i] unless feature T031, "BOOLEAN data type", is implemented. Therefore, the committee responsible for the development of the SQL/MM spatial standard did not want to impose the requirement of Boolean return values.

The method *ST_Distance* exists in addition to the above listed comparison function to quantify the spatial relationship of two geometries according to their distance. The method takes two geometries and the identifier of a unit of measure as input parameter. The distance is measured between all points in the first geometry to all points in the second geometry and the smallest distance is returned as result. Thus, if the two geometries touch or even intersect, the distance between both is intuitively zero.

Generate New Geometries

The fourth set of methods allows the user to generate new geometries from existing ones. A newly generated geometry can be the result of a set operation on the set of points represented by each geometry, or it can be calculated by some algorithm applied to a single geometry. The following methods are examples for both.

ST_Buffer generates a buffer at a specific distance around the given geometry.

ST_ConvexHull computes the convex hull for a geometry.

ST_Difference, ST_Intersection, and ST_Union construct the difference, intersection, or union between the point sets defined by two geometries.

2.3.4 Supporting Data Types

Clause 11 of the SQL/MM spatial standard lists the infrastructure to support angles and directions. To that end, the two types ST_Angle and ST_Direction are defined together with a set of methods. The type ST_SpatialRefSys encapsulates spatial reference systems in a data type. It is introduced in clause 12 and provides an object-oriented interface to spatial reference systems.

Angles and Directions

An angle is used to measure the degree of separation of two intersecting line segments. A type ST_Angle is used to represent angles. The standard does not imply any directionality or rotation (clockwise or counterclockwise) of an angle that could be indicated by a sign.

A multitude of constructor and conversion methods are specified. One can construct an angle from any of the following:

- a numeric radians, degrees, or gradians value,
- a string containing the degrees, minutes, and seconds (DMS),
- three points defining two line segments,
- two directions,
- two linestrings, or
- a well-known text representation.

An angle can also be converted to any of these formats. Additionally, a whole set of mathematical operators to add, subtract, multiply, or divide angles was included.

The concept of directions in the standard is based on angles. A direction is expressed as azimuth or bearing. Azimuth is the horizontal component of a direction. It is measured measured clockwise around the horizon. It is usually measured in degrees from the North direction. A bearing is very similar (sometimes considered to be identical) to an azimuth. A set of methods on the type ST_Direction provides the necessary conversions.

Likewise to angles, directions can be constructed from several different source formats:

- radians,
- indicator for north or south, an angle, and an indicator for east and west,
- an indicator for north and south and an angle,

- two points,
- a linestring, or
- a well-known text representation.

The majority of the methods on that type is concerned with the conversion to and from those data formats. A direction can also be changed by adding or subtracting angles from the direction.

Spatial Reference Systems

The information about all spatial reference system (SRS) can be found in the view ST_SPATIAL_REFERENCE_SYSTEMS in the Information Schema, which we explained in Section 2.3.5. Extending that mechanism, the type ST_SpatialRefSys is part of the standard to represent a spatial reference system as a single, scalar value.

A spatial reference system always has a unique numerical identifier. This identifier communicates to the spatial methods which SRS shall be used, for example, to construct a new geometry. The second part of an SRS is the actual definition of the coordinate system. A coordinate system is comprised a datum, the prime meridian and a unit of measure. If the coordinate system is a projected one, an additional projection algorithm along with its parameters, and a unit of measure for the projected system may be present. A datum is a spheroid that defines the approximated shape of the Earth.

2.3.5 Spatial Information Schema

The SQL/MM spatial standard defines an Information Schema that provides applications with a mechanism to determine the supported and available spatial features. The Information Schema consists of four views, which are explained in the following.

Originally, the SQL/MM spatial information schema was inherited from the OpenGIS Simple Features Specification for SQL. The OGC specification used and still uses the views or tables (the notion is not clear) GEOMETRY_COLUMNS and SPATIAL_REF_SYS with a different set of columns and different semantics of the data shown in the view compared to the SQL/MM spatial information schema. Inconsistencies and problems in this definition compared to the established SQL information schema led to a complete redefinition in the second version of the SQL/MM spatial standard.

The spatial information schema in SQL/MM lists the spatial columns (also called geometry columns), the supported spatial reference systems, the units of measure, and the implementation-defined meta-variables. The entity-relationship (E/R) diagram [Che76] in *Figure 2.8* shows those views and also their relationship to the views defined in the Information Schema in [ISO03g]. The exact definition can be found in Appendix A.

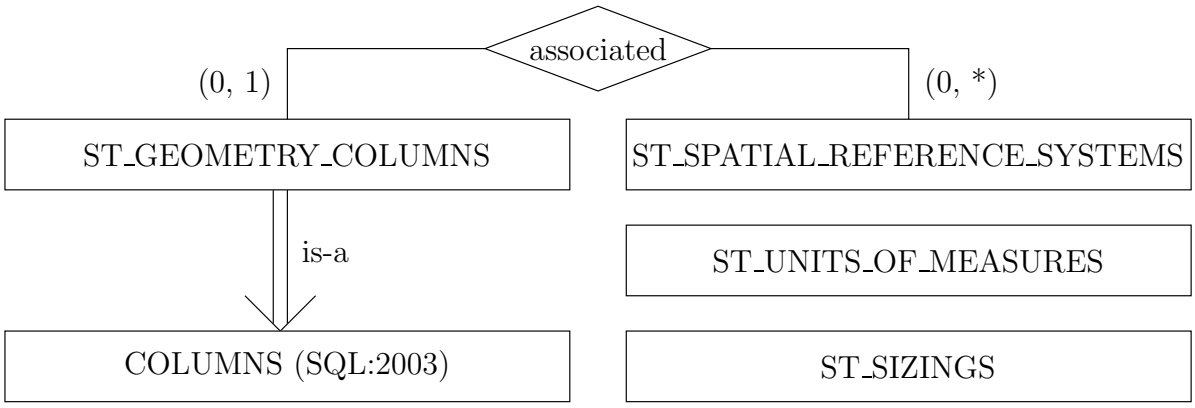


Figure 2.8: Spatial information schema

ST_GEOMETRY_COLUMNS

The view gathers all columns in all tables that have a declared type of ST_Geometry or one of its subtypes. It is not necessary to associate a specific spatial reference system with a column, but an application can choose to do so in order to enforce a respective constraint on the database level. Typically, GISs make use of this feature. The view shows the column identifier, consisting of catalog, schema, table, and column name, and the identifying name and the numeric identifier of the SRS associated with the column.

Appendix A.4 shows that the view COLUMNS from the SQL Information Schema is queried to retrieve the information about all existing spatial columns and then merged with the SRS information for each of the columns that has an associated spatial reference system using an outer join. The spatial columns are identified by their declared types, which has to be ST_Geometry or one of its proper subtypes. The additional information about the SRS for a column is kept in a table of the spatial definition schema, which can also be found in the SQL/MM spatial standard. Its implementation is not mandatory, however, and a product could adopt other means to provide the necessary data for the information schema.

ST_SPATIAL_REFERENCE_SYSTEMS

A spatial reference system has two unique identifiers: a name and a numeric identifier. The name is used in the same way as for all other SQL objects like schemata, functions, or columns. The purpose of the numeric identifier is to pass it to a spatial method that requires the SRS information as input. For instance, a geometry can be constructed in a specific SRS or it can be transformed to another SRS. The numeric identifiers are not very "SQL-like", but they were inherited from the OpenGIS Simple Features Specification for SQL and will remain for backward compatibility and interoperability with OGC.

Along with the identifiers, the view represents the organization (e. g. "EPSG" for the European Petrol Survey Group) that defined this spatial reference system together with the identifier assigned by that organization and the actual definition of the SRS. The definition shows the WKT representation of the SRS.

ST_UNITS_OF_MEASURE

Different units can be used to calculate distances between geometries, the length of curves, the area of surfaces, or to create a buffer around a geometry. For example, a unit of "KILOMETER" can be used. The view lists those units that are supported. An identifying name, the type of the unit (angular or linear), and the conversion factor to the base unit is stored. The base unit is a standardized reference for the respective type of units and its conversion factor is always 1 (one).

ST_SIZINGS

This view contains the spatial-specific meta-variables and their values. An example of a meta-variable is the maximum possible length that can be used for a WKT representation of a geometry, i. e. the variable *ST_MaxGeometryAsText*.

The view ST_SIZINGS has the same intention as the view SIZINGS in the SQL Information Schema of [ISO03g], only specialized for the facilities in the SQL/MM spatial standard. If the SQL standard offered a mechanism for implementations of other standards – including SQL/MM – to add new entries to its view, then ST_SIZINGS becomes obsolete and could be removed from the SQL/MM spatial standard. Unfortunately, the SQL standard does not encompass the requested mechanisms, so both views are necessary.

2.3.6 Sample Usage Scenarios

To illustrate the functionality provided by the SQL/MM spatial standard, this section presents two simple user scenarios. The first scenario is placed in the insurance business. It demonstrates how an enterprise could determine the premiums for an insurance, based on environmental considerations. The second scenario describes how a bank can manage its customers and make decisions on the placement of new branches. Many other applications beyond the traditional scope of geographic information systems (GISs) are possible. A more comprehensive, but by no means all-encompassing, example is given in [Büh04] where a digital library is enhanced to support location-based queries of historical data in northern Germany.

Insurance Company

After a recent flooding, an insurance company wants to correct the information about insured buildings that are in the flood zone and pose an increased risk for the company. The database contains a table RIVERS that contains the rivers and their flood zones and another table BUILDINGS with the data for the buildings of all the policy holders as *Listing 2.1* shows.

```
rivers ( name, water_amount, river_line, flood_zones )
buildings ( customer_name, street, city, zip, ground_plot )
```

Listing 2.1: Simplified relational schema for an insurance company

The column RIVER_LINE contain the linestrings that represent all the rivers in the country. Related to that, the column FLOOD_ZONES shows the flood zones for each river. The ground plot of the customer's building is stored in the column GROUND_PLOT. The tables for the relational model can be created with the SQL statements in *Listing 2.2*.

```
CREATE TABLE rivers (
   name           VARCHAR(30)  PRIMARY KEY,
   water_amount   DOUBLE PRECISION,
   river_line     ST_LineString,
   flood_zones    ST_MultiPolygon )

CREATE TABLE buildings (
   customer_name    VARCHAR(50)  PRIMARY KEY,
   street           VARCHAR(50),
   city             VARCHAR(20),
   zip              VARCHAR(10),
   ground_plot      ST_Polygon )
```

Listing 2.2: Tables for an insurance company database

The first task is to update the information about the flood zones. The flood zones for the river "FLOOD" is to be extended by two kilometers in each direction. The method *ST_Buffer* is used in the following SQL statement for the enlargement of the flood zone. Of course, in a real situation the flood zone would be calculated more precisely. The given statement is meant for illustration purposes only.

```
UPDATE rivers
SET    flood_zones = flood_zones.ST_Buffer(2, 'KILOMETER')
WHERE  name = 'FLOOD'
```

Listing 2.3: Increasing the flood zones of a river

In the next step, the company wants to find all the customers that are in the now-extended flood zone for the river. An SQL statement involving the spatial method *ST_Within* can be used to find all those buildings as *Listing 2.4* illustrates.

```
SELECT customer_name, street, city, zip
FROM   buildings AS b, rivers AS r
WHERE  b.ground_plot.ST_Within(r.flood_zones) = 1
```

Listing 2.4: Getting addresses of customers in flood zones

The so-retrieved addresses can be further processed, and the customers can be informed of any changes to their policy. For example, requirements to raise the houses above the ground level or increases to the premiums could be imposed.

Banking

A bank manages its customers and branches. Each customer can have one or more accounts, and each account is managed by a branch of the bank. To improve the quality of services, the bank performs an analysis of its customers, which also involves a spatial component, the locations of the customer's homes and the branches. The tables in the bank database are defined as in *Listing 2.5*.

```
CREATE TABLE customers (
   customer_id  INTEGER
                   PRIMARY KEY,
   name         VARCHAR(20),
   street       VARCHAR(25),
   city         VARCHAR(10),
   state        VARCHAR(2),
   zip          VARCHAR(5),
   type         VARCHAR(10),
   location     ST_Point )

CREATE TABLE branches (
   branch_id  INTEGER
                  PRIMARY KEY,
   name       VARCHAR(12),
   manager    VARCHAR(20),
   street     VARCHAR(20),
   city       VARCHAR(10),
   state      VARCHAR(2),
   zip        VARCHAR(5),
   location   ST_Point,
   zone       ST_Polygon )

CREATE TABLE accounts (
   account_id   INTEGER  PRIMARY KEY,
   routing_no   INTEGER  NOT NULL,
   customer_id  INTEGER  NOT NULL,
   branch_id    INTEGER  NOT NULL,
   type         VARCHAR(10)  NOT NULL,
   balance      DECIMAL(14, 2)  NOT NULL,
   FOREIGN KEY(customer_id) REFERENCES customers,
   FOREIGN KEY(branch_id) REFERENCES branches )
```

Listing 2.5: Tables of a database of a bank

The first query determines all customers with an account balance larger than $10,000.- in any of the accounts and who live more than 20 miles away from their respective branch. Profitable customers can be found in order to serve them better with a new branch with that information.

```
SELECT DISTINCT c.customer_id, c.name
FROM   customers AS c JOIN accounts AS a ON
          ( c.customer_id = a.customer_id )
WHERE  a.balance > 10000 AND
       a.location.ST_Distance(
          ( SELECT b.location FROM branches
            WHERE  b.branch_id = a.branch_id ),
          'MILES') > 20
```

Listing 2.6: Finding profitable customers

Each branch has an associated target sales zone. The bank wants to find all the portions of the assigned sales zones of the branches that overlap. It is not intended to have more than one branch assigned to a certain area, and any duplicates are to be found and corrected so that competition between branches can be reduced in the future. The query in *Listing 2.7* retrieves the identifiers for each two branches that have an overlap in the zones and also the overlapping area, encoded in the well-known text representation.

```
SELECT b1.branch_id, b2.branch_id,
       b1.zone.ST_Intersection(b2.zone).ST_AsText()
FROM   branches AS b1 JOIN branches AS b2 ON
          ( b1.branch_id < b2.branch_id )
WHERE  b1.zone.ST_Overlaps(b2.zone) = 1
```

Listing 2.7: Detecting overlapping sales zones

As a customer service, the bank wants to find all the customers that live within a 10 miles radius of a branch that does not manage their accounts. The accounts are to be transferred to a closer branch if the customer agrees. *Listing 2.8* shows an SQL query that can be used for that purpose. The complexity of the statement does not lie in the handling of the spatial information, but rather in determining the current branch associated to a customer in the subselect.

```
SELECT c.name, c.phone, b.branch_id
FROM   branches AS b, customers AS c
WHERE  b.location.ST_Buffer(10, 'MILES').
          ST_Contains(c.location) = 1 AND
       NOT EXISTS (
          SELECT * FROM accounts AS a
          WHERE  a.customer_id = c.customer_id AND
                 a.branch_id = b.branch_id )
```

Listing 2.8: Finding closer branches for customers

2.3.7 Discussion of the Standard

The SQL/MM spatial standard provides a rich set of types, methods, and functions. However, it cannot be denied that some additions, changes, and even omissions would result in a further improvement of the standard. The issues that we found are outlined now, along with suggestions for a correction.

The spatial type hierarchy cannot be seen as an optimal way to model spatial information in a relational database system. The OGC class hierarchy was directly carried over to the SQL/MM spatial types. The set-oriented functionality of SQL was not considered. Additionally, existing problems from the OGC class hierarchy were inherited. For example, the empty geometries are not properly represented but rather mixed into all data types. The standardized spatial type hierarchy can be improved by a different alignment of the inheritance rules between the types. Such a modification has a significant impact on the standard. It merits its own discussion and we specifically dedicated Section 2.3.8 to that.

Methods on the Geometry Types

The standard defines constructor methods that can handle well-known text and well-known binary representations. It does not allow for a handling of the GML representation in a constructor method, however. The existing constructor methods for the WKT representation could be reused for that, given that WKT and GML are both a textual representation for geometries and can easily be distinguished be analyzing the very first non-whitespace character. If it is a `'<'`, it must be the first character of an opening XML tag and that only occurs in the GML representation. With the constructors handling GML, the functions like *ST_PolyFromGML* could then be removed. Additionally, they are not defined in the OpenGIS Simple Features Specification for SQL so that no compatibility issues arise.

A set of functions allows the user to construct any geometry using one of the external data formats. Those functions act like factory functions and are named *ST_GeomFromText*, *ST_GeomFromWKB*, and *ST_GeomFromGML*. Instead of using the explicit names to denote the format handled by each function, an approach similar to the constructor methods is preferable. An overloaded function *ST_Geometry* can be defined with the same semantical behavior as the existing functions. Constructor methods on the type ST_Geometry cannot be used because that type is not instantiable.

The two methods *ST_AsText* and *ST_AsBinary* have not very well chosen names because there are many different textual and binary formats to represent geometries. Names like *ST_AsWKT* and *ST_AsWKB* would be more appropriate as they clearly indicate the target format for the conversion. That argument was already taken into consideration for the *ST_AsGML* method, whose addition to the standard encountered the issue.

There are several groups of methods that provide (nearly) identical functionality. For example, the methods *ST_Intersects*, *ST_Crosses*, and *ST_Overlaps* all test for intersections of the interiors of the two input geometries. The only difference between the methods is that *ST_Crosses* does not allow to test if a surface intersects some other geometry, or if some other geometry intersects a point. *ST_Overlaps* requires that both geometries to be compared have the same dimension. For example, a line can overlap another line but not a polygon. If the conditions are not met, the result is NULL. *ST_Intersects* is the generalized version of the functionality to test for the overlay of geometries. It does not impose any restrictions in its input parameters. The existence of *ST_Crosses* and *ST_Overlaps* is rather questionable.

Another example of duplicated functionality is the method *ST_GeometryType*. It returns the name of the most-specific type of a spatial value. The function *TYPE_NAME* in SQL:2003 [ISO03i] provides the same information, making the spatial method obsolete.

An omission can be found in the support for external data formats. The de-facto industry standard to represent geometries is the so-called ESRI shape format [ESR98]. Even if the shape format comes with a few shortcomings with respect to distinguishing the geometry types, it would be desirable to incorporate it into the SQL/MM spatial standard. However, the copyright for that format is held by the Environmental Systems Research Institute (ESRI) and that fact prohibits a general standardization. Another format whose importance is growing steadily are Scalable Vector Graphics (SVG) [W3C03]. SVG was originally intended for any kind of vector graphics. Maps generated from spatial data are instance of vector graphic images. The integration of spatial databases with web browsers can easily be achieved with SVG because most modern browsers are able to render SVG files. That means that web-based applications can be extended to incorporate spatial information. Furthermore, light-weight web-based GIS become possible.

The current SQL/MM spatial standard reached a stable point in terms of the functionality it defines. However, additional functionality to support more spatial oriented logic should be included in a future version. For example, methods to modify geometries directly in the database system can be added. There are no simple functions to change a point of a linestring, or to generalize geometries if it is too detailed, i. e. too many points are used to define it. Therefore, applications have to retrieve a geometry from the database management system (DBMS), construct a spatial object, modify this object and pass it back to the DBMS. A simple method invocation at the database level could simplify the whole process. Existing products already support methods like *ST_ChangeVertex* or *ST_Generalize* [IBM04d].

Angles & Directions

The standard defines a framework for angles and directions that is very complex and rather difficult to use. The best example are the *ST_Add*, *ST_Subtract*, *ST_Multiply*,

and *ST_Divide* methods for the ST_Angle type. It would be preferable to just use the operators +, -, *, and /, respectively.

That a much simpler approach could be implemented is apparent when looking at the way how the standard defines both types. Each of the types has only a single attribute. ST_Angle uses an attribute of type DOUBLE PRECISION, and ST_Direction uses an attribute of type ST_Angle. Instead of wrapping a simple DOUBLE PRECISION into the types directly or indirectly, the DOUBLE PRECISION could be provided to the user, together with a set of functions to convert the floating point number it represents to the various other formats. Even the strong typing can be retained if a distinct type is used.

A possible implementation of the same functionality that is currently defined in the standard to support angles can be achieved with the functions shown in *Listing 2.9*. The first set of functions constructs an angle from the given input and returns it as radians (DOUBLE PRECISION). The second set of functions can then be used to convert the angle (in radians) to another representation.

```
ST_Angle(unit CHARACTER(1), angle DOUBLE PRECISION)
ST_Angle(degrees INTEGER, minutes INTEGER,
         seconds DOUBLE PRECISION)
ST_Angle(pt1 ST_Point, pt2 ST_Point, pt3 ST_Point)
ST_Angle(line1 ST_LineString, line2 ST_LineString)
ST_Angle(wkt CHARACTER VARYING(ST_MaxAngleAsText))

ST_Degrees(radians DOUBLE PRECISION)
ST_Gradians(radians DOUBLE PRECISION)
ST_DegreeComponent(radians DOUBLE PRECISION)
ST_MinuteComponent(radians DOUBLE PRECISION)
ST_SecondComponent(radians DOUBLE PRECISION)
ST_AsText(radians DOUBLE PRECISION)
```

Listing 2.9: Functions to support angles and directions

For completeness, a conversion from gradians or DMS to radians should also be implemented. Existing database systems support such conversion routines as built-in functions today, completely independent of the respective spatial products. For example, DB2 offers the functions *DEGREES* and *RADIANS* to convert radians to degrees and vice versa [IBM04e].

Spatial Reference Systems

The type ST_SpatialRefSys is completely detached from all other spatial types and methods. Therefore, the very existence of the type is questionable. Values of that type are not used as input parameter for any of the routines that produce new geometries, nor

are they returned as result from any of the spatial methods. Additionally, all the information about spatial reference systems is stored and maintained in the Information Schema already.

The only justification for the existence of the type is – besides the conversion to and from the WKT representation – the associated method *ST_Equals*, which allows the user to compare two SRSs. However, according to Clause 16 of the SQL/MM spatial standard, which specifies the conformance rules, it is not mandatory to create the type ST_SpatialRefSys. A simpler alternative would be a function *ST_EqualSRS* that receives the identifiers of the two SRS that are to be compared. The DB2 Spatial Extender provides such an interface with the functions *ST_EqualCoordSys* and *ST_EqualSRS* [IBM04d]. The two different functions stem from an extended implementation for spatial reference systems in the DB2 Spatial Extender, which adds the notion of coordinate systems as Section 2.5.1 explains.

Information Schema

The view ST_SPATIAL_REFERENCE_SYSTEMS in the spatial information schema uses a very simplified way to manage spatial reference systems. The European Petrol Survey Group (EPSG), which has reformed as the Surveying and Positioning Committee of the International Association of Oil and Gas Producers (OGP), developed a more expressive schema for spatial reference systems [EPS04], which is also described in the ISO 19111 standard [ISO03b], although in a different context. This schema, the EPSG geodetic parameter dataset, is illustrated in *Figure 2.9*. The figure shows a total of 15 tables. Some of the tables contain different types of entries, for example spherical, ellipsoid, affine and polar coordinate systems are collected in the table Coordinate System. The schema distinguishes between spherical, ellipsoidal, Cartesian, affine, polar, gravitation-related, cylindrical, and linear coordinate systems. Likewise, coordinate reference systems are either geocentric, geographic-3D, geographic-2D, projected, vertical, or engineering-specific[3]. There can be geodetic, vertical, or engineering-specific datums, and the table Coordinate Operation stores information about concatenated or single coordinate operations, and single operations are either transformations or conversions. Thus, a total of 32 different information is stored in the 15 tables.

Such a fine grained and application-specific modeling of SRS might reach well beyond the scope of the general purpose spatial data processing. However, it is usually necessary for very GIS-centric applications. The information from all the 15 tables in the EPSG schema are condensed into a singe DEFINITION attribute in the catalog view ST_SPATIAL_REFERENCE_SYSTEMS in the SQL/MM spatial standard. Both interfaces have their justification, so the more structured way of the EPSG schema should be provided in addition to the current information schema.

[3]Engineering-specific coordinate reference systems are used in CAD/CAM applications, for instance.

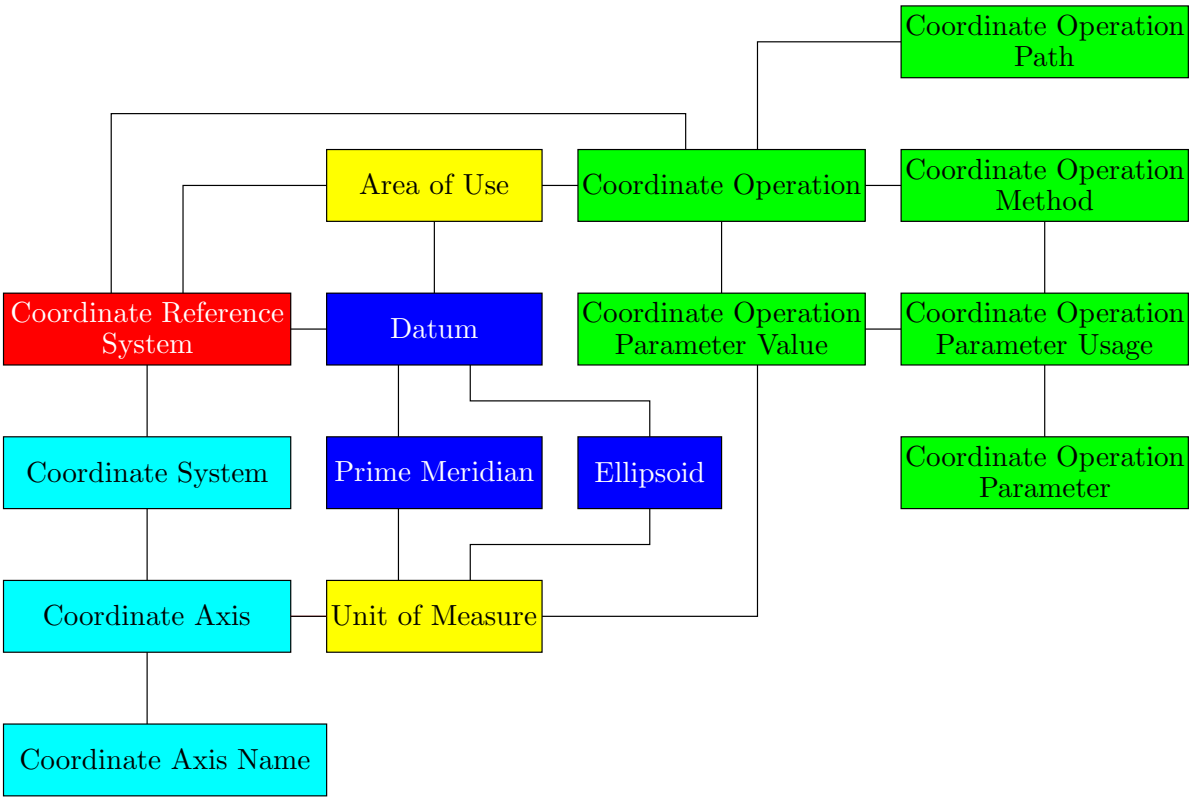


Figure 2.9: EPSG schema for spatial reference systems

2.3.8 Improved Spatial Type Hierarchy

An analysis of the type hierarchy as defined by the SQL/MM spatial standard raises several questions on the design decisions made. The type hierarchy, as derived from the OGC class hierarchy and extended, is clearly targeted towards inheriting the implementation-specific details instead of focusing on the usability of the data types from a user perspective. This flaw can be found in many different applications [DD00] and it is due to the simple transfer of concepts from a procedural, navigational programming language to the set-oriented, relational model of SQL.

We aim at addressing the problems in the SQL/MM spatial type hierarchy. It describes the existing major issues and proposes solutions for each. Some of the problems were already described in [Sto03b] and a modified spatial type hierarchy is presented. The feasibility and benefits of such a different type hierarchy was demonstrated in [Gol06a], which used the DB2 Spatial Extender [IBM04d] and reimplemented the complete hierarchy and attached the original spatial methods to the proper types in the new hierarchy. As a result, none of the methods exists in multiple branches of the hierarchy, e. g. *ST_Length*. Additionally, conversions from geometric primitives to collections are implicitly provided through the inheritance rules.

Empty Geometries

The type hierarchy does not address the concept of empty geometries in form of a separate type under ST_Geometry (cf. Section 2.3.2). An empty geometry represents an empty set of points. Such a geometry results from the intersection of two disjoint polygons, for example. Salomon [Sal06a] describes other possible applications for empty geometries for high-performance filtering of geometries.

The SQL/MM spatial standard allows that a value of each of the instantiable types can be an empty geometry. Thus, empty points, empty linestrings, empty polygons, and empty geometry collections etc. exist. If a method has to return an empty geometry as its result and the most specific type is not inherent from the method definition, then an empty point is generated. Implicit or explicit casts can be used to convert empty geometries from one type to another.

Three different approaches can be chosen to model empty geometries:

1. use NULL,
2. use values of a dedicated type under ST_Geometry, or
3. allow that a value of an instantiable type can represent an empty geometry.

The third option was derived from the OGC specification and implemented in the standard. It raises the question what, for example, an empty polygon is supposed to be. Either the geometry is a polygon, i. e. it has an enclosed area and a boundary, or it is empty. Treating both as polygons is not very intuitive.

The notion to use NULL for empty geometries is very attractive. NULL is untyped and, thus, can be used as input parameter for any of the spatial methods on any of the geometry types. The main argument against NULL is a semantical one. NULL is usually used to represent *non-existing*, *unknown*, or *not applicable* data. In the case of empty geometries, those concepts do not apply. For example, the results of the method *ST_Intersection* are very well defined and known. The resulting geometry of such an operation might just be the empty set, which is different from an *unknown* result. Of course, the point of a simpler ease-of-use cannot be denied.

Using a separate type, like ST_Empty, to model empty geometries was not acceptable to the standards committee with the reasoning that all of the methods that are only defined on proper subtypes of ST_Geometry would have to be defined on ST_Empty again. However, this argument is void because an observation of the methods defined on the proper subtypes of ST_Geometry reveals that none of those methods is applicable for empty geometries. Those are specific methods for specific types of geometries.

Another concern was that some methods that return values of a proper subtype of ST_Geometry can also return an empty geometry. It might not be possible for those

methods to return empty geometries anymore, unless the return type would be changed to ST_Geometry, which is incompatible to previous versions of the standard. An analysis of all the methods defined in the standard reveals that each method either returns values of type ST_Geometry, a non-spatial type like INTEGER, or that empty geometries cannot appear as results if empty geometries are consistently represented with a dedicated type. Thus, the existing methods do not impose an obstacle either.

A major reason for the standards committee to not use a separate type in the type hierarchy was the goal to align the ISO standard as much as possible with the corresponding OGC specification. Thus, the currently defined approach was adopted and much work was invested to clarify the behavior of the methods if they receive an empty geometry as input. User-defined casts to convert an empty point to an empty polygon or to any other instantiable type were also added for this purpose. All this work would not have been necessary with a more proper modeling of empty geometries using a type like ST_Empty instead. The type hierarchy would also be more intuitive.

Improper Representation of Geometry Collections

An important goal of an SQL/MM spatial standard should be the usability of the functionality it defines. SQL is a set-oriented language, and iterating over the elements of a collection as mandated is, although possible, not that simple in SQL. A dynamic compound statement with procedural logic or a recursive subquery has to be employed for such a task. The respective method on the element of a collection has to be invoked during the iteration, and the results are to be combined. Leaving such a task up to the user will only lead to the user defining those functions himself to prevent the repetition of the logic, making its standardization attractive. Thus, a better approach would be to avoid the need for iterative processing in the first place.

Furthermore, the fine grained type system adds another layer of complexity if the results of a spatial operation are to be processed further. For example, the SQL query in *Listing 2.10* is supposed to return all the parts of the geometries in column SPATIAL_COLUMN that fall into a certain rectangle. The rectangle in question is represented using a polygon. This is a very typical query for spatial database systems to find all geometries that fall into the area and clip them accordingly.

```
SELECT spatial_column.ST_Intersection(
          ST_Polygon('polygon((10 10, 10 20, 20 20,
                               20 10, 10 10))'))
FROM   spatial_table
WHERE  spatial_column.ST_Intersects(
          ST_Polygon('polygon((10 10, 10 20, 20 20,
                               20 10, 10 10))')) = 1
```

Listing 2.10: Querying a spatial column

Assuming that the column contains values of type ST_MultiPoint, the results of the query can contain any of the following for each row:

- an empty geometry (empty point),
- a single point, or
- multiple points (multi-point).

In the above query, the results can be retrieved and visualized on the screen or plotted on paper as map. Further processing the results in an extended SQL statement is, however, not completely trivial. Because ST_Point and ST_MultiPoint are in two independent subtrees of the type hierarchy, only the methods available on the common ancestor ST_Geometry can be used directly. A conversion of the single points to multi-points with only a single element is not supported by the standard, instead only the conversion of empty geometries from one type to another is possible. The only other alternative lies in using `CASE` and `TREAT` expressions. Those can be used to detect the most-specific dynamic type of each value and then invoke more specific methods depending on that type. Clearly, the complexity of this approach grows exponentially if several spatial methods are chained that way.

A well-known rule for designing object-oriented classes states that if there are two classes with the same or very similar methods, then those classes are most probably very similar as well and should be combined into a single class [Bal00]. The SQL/MM spatial type hierarchy does not follow this rule. Instead, methods like *ST_IsClosed* or *ST_Length* are defined for ST_Curve and ST_MultiCurve, both.

Another ambiguity exists between the types ST_CompoundCurve and ST_MultiCurve. The very same geometry, a compound curve, can be represented as multi-curve or as geometric primitive using the ST_CompoundCurve type. The semantical difference between both types is that a multi-curve can represent a set of disconnected curves wheres a primitive curve is always connected. Therefore, both types could and should be combined into a single type, and a new method like *ST_IsConnected* could be used to extract the property whether all parts of the multi-curve are connected, effectively giving the same information as compound curves already provide today.

A solution to address all those issues is to set the types ST_Point and ST_MultiPoint in direct relationship by using inheritance. Interpreting this paradigm in the context of the SQL type hierarchy gives: ST_MultiPoint is a specialized ST_GeomCollection, which only contains points, and ST_Point is a specialized ST_MultiPoint, consisting of only a single point. A similar approach can be implemented for the other types, which are not in the subtree under ST_GeomCollection. To this end, the type ST_GeomCollection is not needed as it can be merged into ST_Geometry *Figure 2.10* shows how such a type hierarchy could be defined. Additionally, the new type hierarchy adds a type to represent empty geometries. None of the data types will be non-instantiable as they are either geometric primitives or homogeneous and heterogeneous collections.

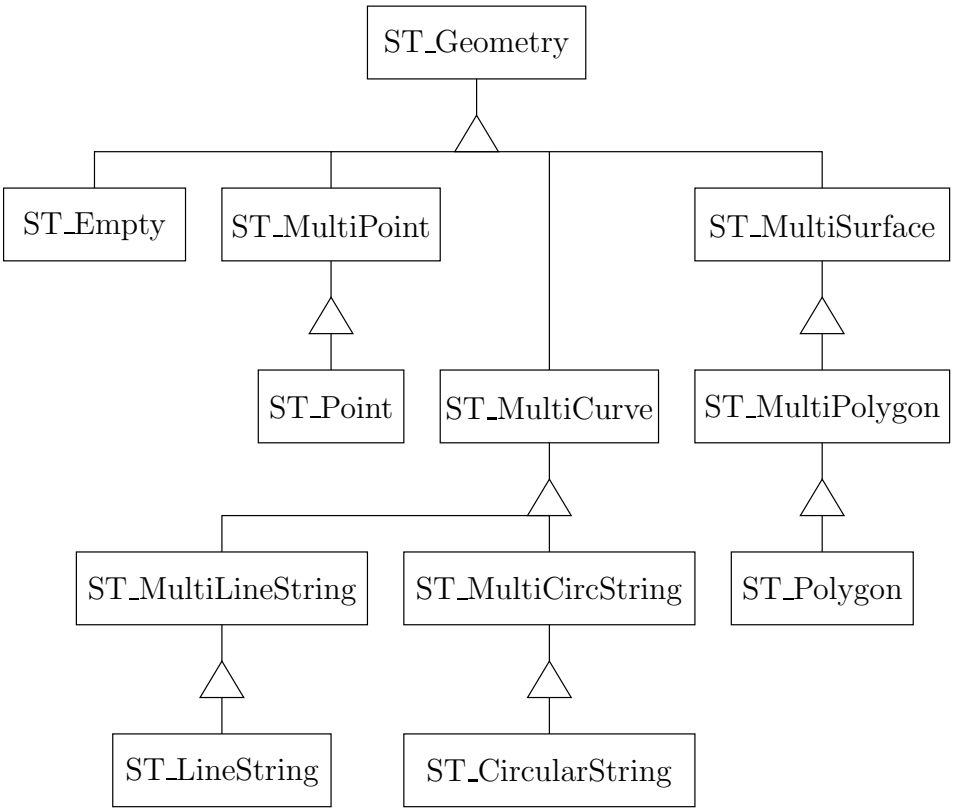


Figure 2.10: Modified spatial type hierarchy

A similar idea was apparently the basis for the type hierarchy implemented in MapInfo for Microsoft SQL Server [Map04]. Other products follow the type hierarchy of the SQL/MM spatial standard. We give a more detailed overview on the products that provide spatial extensions for relational database management systems (RDBMSs) and their type system in Section 2.5.

2.3.9 Major Gaps in the Standard

Applications that exploit the functionality of spatial database systems are often written in a programming language that is quite different from SQL. These days, the Java programming language [GJSB05] is very common. Although the SQL/MM spatial standard is based on the structured types as they are defined in the SQL standard itself [ISO03i], there is no adequate mechanism available to transport geometries between the database system and the application logic in an obvious manner. Chapter 3 illustrates how the spatial data could be made accessible in a Java application in a straight-forward manner. There, we also discuss which facilities are provided by the SQL standard and why they do not fit very well for spatial applications.

Another important aspect is the support for network or graph-based functionality. To that end, the Japanese standards committee supplied a change proposal in the year 2002 that adds a function *ST_ShortestPath*, which calculates the shortest path in a network derived from of linestrings between two given points. The proposal was rather brief in the semantics of the new function. For example, it did not explain how the network (or graph) is to be built from the linestrings, whether it will be a directed or undirected graph, and how intersections must be represented. The proposal also ignored the fact that a real implementation will require some kind of materialization in order to return the results with an acceptable performance. Chapter 4 describes the concepts of an integrated graph management for spatial applications. It is tailored to support routing operations in a network built from spatial data or to construct the dual graph for a space partitioning defined by polygons or other topological information. A prototypical implementation demonstrates the functional feasibility of those concepts and presents performance results.

Finally, some future directions of the SQL/MM spatial standard already show up in the current working draft with the addition of Z and M coordinates to the geometries. Although these hints at the management of three-dimensional data, the standard (or the current working draft) has not yet reached that stage. Today, only points, lines, and polygons as well as collections thereof can be represented in $\mathbb{R}^2$ or $\mathbb{R}^3$ and even $\mathbb{R}^4$ with the M coordinates. Those are all zero-, one- or two-dimensional objects only, and there are no means to represent solids, for example tetrahedra, in the spatial type hierarchy. Chapter 5 is dedicated to the seamless extension of the type hierarchy to support such geometries. The overall impact of such an extension on the SQL/MM spatial standard is discussed, and an implementation proves those concepts.

2.4 Geography Markup Language

The previous section introduced the SQL/MM spatial standard. The roots of that standard can be found in the SFS [OGC99]. Since the initial development of SFS, the OGC published the latest Version 1.1 in the year 1999 and since then no new work was incorporated. The background for the lack of attention is that the focus of the OGC switched to the Geography Markup Language (GML) and a new Implementation Specification was developed based on eXtensible Markup Language (XML) emerging at this time. Today, the GML specification [OGC04] is already in its third revision, and the OGC is working closely with the International Organization for Standardization (ISO) to transform the GML specification into an international standard [ISO05b].

The GML specification is mainly concerned with the management and representation of geographic features, including all the attributes that might be applicable to certain GIS-specific markets like transportation or Earth surveying. A small portion of the GML specification is still concerned with the geometries themselves, but a significant

part goes way beyond the scope of the SQL/MM spatial standard. Given the history of OGC's specification, it is understandable that GML contains a further developed type hierarchy. *Figure 2.11* shows that hierarchy, ignoring all attributes that the specific types may have. Also, the relationships between all the various types are not depicted. We only provide the inheritance information as it is relevant to SQL/MM spatial type hierarchy presented in Section 2.3.2.

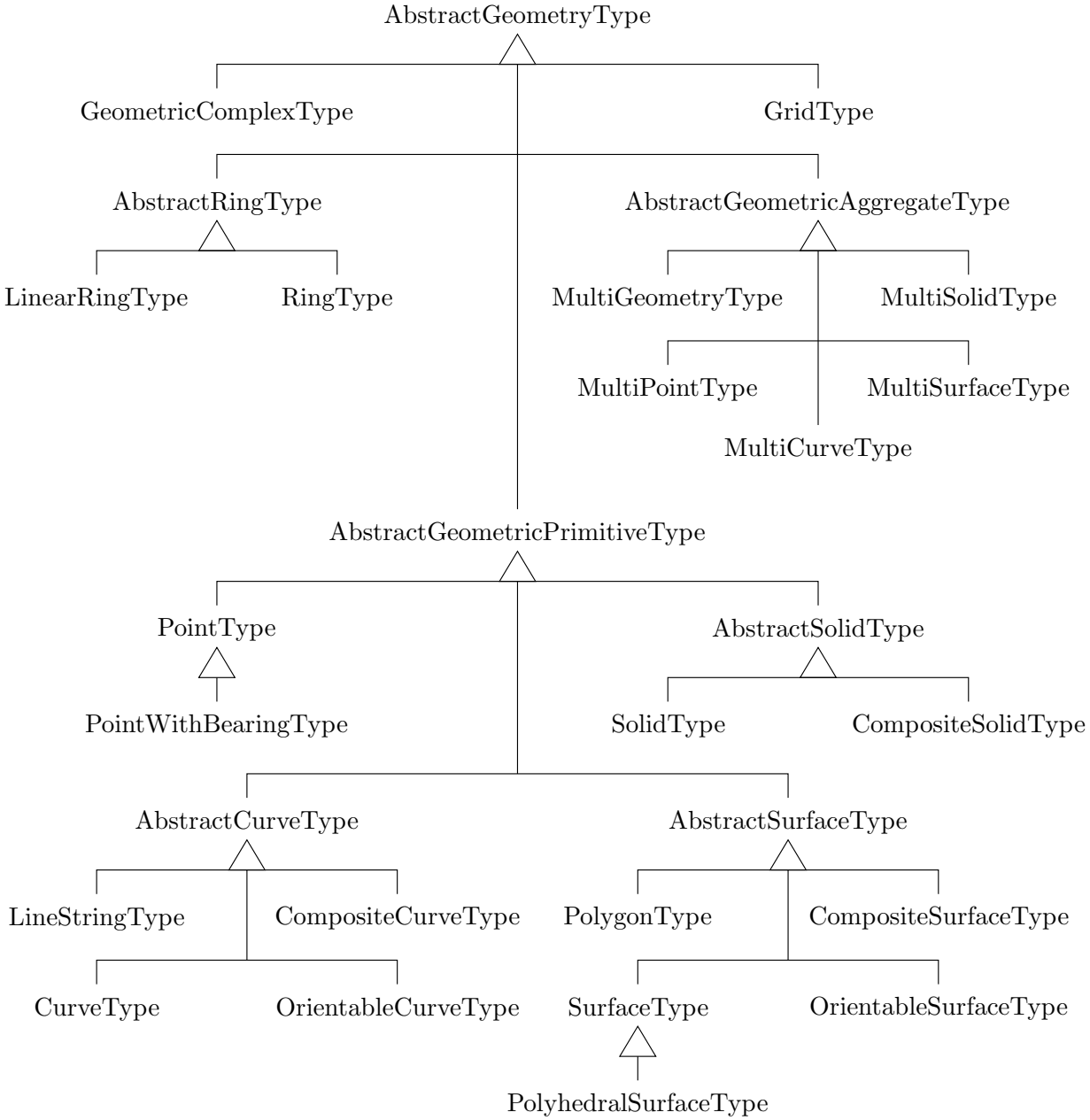


Figure 2.11: GML type hierarchy

The GML type hierarchy is very complex. It also comes with the focus towards implementation of geometric data as opposed to an intuitive modeling. Thus, the type hierarchy carries along the legacy of the OpenGIS Simple Features Specification for SQL and its problems regarding an implementation in a set-oriented language that were discussed earlier. Additionally, new ambiguities were introduced. For example, the type MultiGeometryType is no ancestor of MultiPointType, which implies that a multi-point geometry could be represented by either type. Rings are not considered as curves or linestrings. And a distinction between aggregates, composites, and complex geometries is not existent. A composite curve could be modeled using the type ComplexCurveType, but also using the types GeometricComplexType or MultiCurveType. The property that a composite curve must consist of a set of connected curves is not guaranteed by MultiCurveType, nevertheless this issue begs to question the type hierarchy. Composite curves are just one example, and there are two other composite types (CompositeSurfaceType and MultiSurfaceType) that raise the same questions.

GML and its type hierarchy are not the basis for the following chapters as the GML does not consider spatial operations and their semantics. GML specifies rather a data exchange format for spatial data and its associated attributes. Thus, it is not directly applicable to this thesis. That fact shall not diminish the importance of GML, however. Certain aspects of it – like the approach to model 3D objects – are referred to in the following chapters.

2.5 Products Implementing Spatial Extensions

As spatial data becomes more and more important in today's applications, the pressure to include spatial operations in database systems grows. Many commercial and open source database systems already provide extensions to store spatial data in relational tables. We introduce only the most important of these products here (in an alphabetical order). For each product, we give a short summary of the type system and the associated functionality. We also explain the relation to the SQL/MM spatial standard. Additional products and prototypes have been developed, e. g. [FFS00], but they are not very relevant in practical terms.

2.5.1 IBM DB2 Spatial and Geodetic Extenders

The DB2 Spatial Extender [IBM04d] is intended to provide spatial functionality for DB2 UDB databases. The extender follows the SQL/MM spatial standard very closely. It implements the full standardized information schema, the spatial type hierarchy, the methods and functions applicable to the types, and the transform functions for the external data formats WKT, WKB, and GML. The data types ST_CircularString and ST_CompoundCurve are not provided, but those types are optional according to the

conformance rules of the standard. The DB2 Spatial Extender defines all the methods on the geometry types as is mandated by the standard. Beyond that, many additional methods are implemented to provide additional functionality to users.

The extender manages data in $\mathbb{R}^2$, but Z coordinates and M coordinates (measures) can be added to each geometry. Methods like *ST_Is3D* or *ST_MeasureBetween* can take advantage of the additional information. However, the computation will always be performed in the Cartesian two-dimensional data space or on the two-dimensional surface of a spheroid in case the geodetic feature of the product is employed.

Extending the standardized notion of envelopes for a geometry, the DB2 Spatial Extender implements several methods that work with MBRs (minimum bounding rectangle) of geometries, for example *ST_MBRIntersects*. The difference between MBRs and envelopes applies to points and the special cases of horizontal and vertical linestrings only, where the envelope is a true rectangle whereas the MBR is the point or the linestring itself, i. e. a collapsed rectangle. Along with the MBR, the extender introduced methods to directly retrieve the minimum and maximum coordinate values for all four dimensions, for example *ST_MinX* and *ST_MaxY*.

Further functionality that should be standardized as part of the SQL/MM spatial standard is related to the direct modification of geometries like the addition, deletion, or update of points in linestrings with the methods *ST_AppendPoint*, *ST_RemovePoint*, or *ST_ChangePoint*, respectively. Simplification of very detailed linestrings or polygons can be performed through the method *ST_Generalize*.

Like all other commercial extensions, the extender is developed independently of the DB2 database kernel and is not deeply integrated. It solely relies on DB2's interfaces for the definition of spatial data types, spatial routines, and spatial index mechanisms. Separate performance tuning considerations – especially for the spatial index mechanism – are applicable and discussed in the literature [Sto05a, SA02, SRC02a, SRC02b].

DB2 does not support constructor methods for structured types, so the DB2 Spatial Extender had to work around this issue by providing the constructor methods as functions. For example, the construction of a point value from its WKB representation *wkb* is done with the expression `ST_Point(wkb, srs-id)` instead of `NEW ST_Point(wkb, srs-id)`. From a user perspective, that is just a syntactical issue. In addition to the three standardized representations WKT, WKB, and GML, geometries can be constructed from binary representations that adhere to the ESRI shape format [ESR98]. The constructor functions deal with all four formats and determine the specific format internally from the data and decode it accordingly.

The internal representation is based on integer values for the coordinates, which requires that the floating point values of the external representation are converted. To that end, the spatial reference system, which is associated with the geometry value upon construction, defines offset and scale factors for the conversion between the internal and external representation.

Another feature available as part of the DB2 Spatial Extender is a framework for geocoding functionality [Sto03a]. A geocoder is a function that takes an address as input and returns a point on the real world for that address. More generally, a geocoder receives a set of input parameters and generates a geometry form it. Some sort of reference data might be consulted for the computation, for example a base map detailing streets and house number ranges.

2.5.2 IBM Informix Spatial DataBlade

The Informix Spatial DataBlade Module [IFX02b] is tailored as a plug-in for the Informix Dynamic Server (IDS). It is very similar to the DB2 Spatial Extender introduced in Section 2.5.1. Both products share the same roots. ESRI was involved with the initial development of both products. ESRI also participated in the effort that resulted in the OpenGIS Simple Features Specification for SQL [OGC99]. Therefore, it is no surprise that the Informix Spatial DataBlade defines the standardized spatial type hierarchy and the related functions and methods for the geometry types.

The tables of the information schema are still based on the old and inconsistent version defined in [OGC04]. This schema was also present in the first version of the SQL/MM spatial standard. Due to its inherent problems, it has been replaced in the second version of the standard [SK01]. The Informix Spatial DataBlade does not support constructors that have the same name as the data type for which a new value is generated. Those were added in the second version of the standard and the product did not yet adopt them. The omission does not impose a technical shortcoming, but it can be considered as an inconvenience and makes the usage less obvious. For example, a point is constructed using the functions *ST_PointFromText* or *ST_PointFromWKB*.

The set of routines of the Spatial DataBlade includes several non-standardized functions. The additional methods carry the prefix *SE_* instead of *ST_* to distinguish them from the standardized functionality. Z and M coordinates in geometries are supported. Spatial operations take place in the two-dimensional data space and functions like *ST_Is3D* or *ST_LocateAlong* can use the additional information. Modification operations on geometries are available through the functions *SE_VertexAppend*, *SE_VertexDeleted*, and *SE_VertexUpdate*.

Geometry values can be constructed from their WKT or WKB representation as is mandated by the SQL/MM spatial standard. Additionally, GML fragments can be used as external format and also the ESRI shape format is supported. Likewise, the geometries can be converted to any of those four formats with the functions *ST_AsText*, *ST_AsBinary*, *SE_AsGML*, and *SE_AsShape*, respectively.

Geodetic processing (spatial operations on the surface of an ellipsoid) is also available, but only through another, incompatible product. The Informix Geodetic DataBlade Module [IFX02a] uses different spatial data types and also different methods.

2.5.3 MapInfo SpatialWare

MapInfo Corp. is a major vendor in the GIS market. They defined their own extension SpatialWare for several database management systems, including Microsoft SQL Server and Informix Dynamic Server. There are also plans to port SpatialWare to DB2 UDB in the near future [Map01]. The following description refers to SpatialWare for Microsoft SQL Server [Map04] as no documentation for other database systems is available anymore (e. g. SpatialWare for Informix Dynamic Server) or yet (e. g. SpatialWare for DB2 UDB).

The spatial types that can be handled by SpatialWare differ from the SQL/MM spatial standard and also from the OpenGIS Simple Features Specification for SQL. The product does not define any types for geometry collections, and the type hierarchy is dedicated to geometric primitives. Collections can be stored, however, by using the root type ST_Spatial directly. The types ST_Point and ST_Polygon are available, and linestrings are to be modeled using ST_PolyLine. Additionally, SpatialWare provides the type ST_CircularArc, which corresponds to the standardized type ST_CircularString. A set of non-standardized types is implemented to give the user an increased control over the geometries stored in the database. To that end, the specialized types HG_Box, HG_Quad, HG_Triangle, HG_Circle, and HG_Curve are provided to represent rectangles, quadrangle, triangles, circles and smoothly curved lines.

The complex type hierarchy is not made available through explicit data types in the database system. Instead, the type ST_Spatial has to be used as declared type for spatial columns in any table. The more specific type information is managed internally, and the user has to be aware of it only when a geometry is converted from or to an external format.

The provided spatial functionality includes and exceeds the definitions of the SQL/MM spatial standard. The standardized functions carry usually the prefix *ST_* whereas all other functions are prefixed with *HG_*. However, deviations exist like the function *HG_Is_Closed*, which also exists in the standard. SpatialWare supports X, Y, and Z coordinates for all geometries. Generally, the spatial processing takes place in $\mathbb{R}^2$ ignoring the Z coordinates. Specific functions like *ST_Length_3D* or *HG_Center_In_3D* do exploit the Z coordinate.

Geometries can be constructed in SpatialWare from the WKT or WKB formats. Additionally, so-called geometry strings can be used, which is a proprietary textual representation that lists all the points explicitly using a functional notation. It is similar to nested function calls in SQL syntax as *Listing 2.11* demonstrates for a triangle.

```
'HG_Triangle ( LIST { ST_Point(1, 3), ST_Point(5, 7),
                      ST_Point(9, 11) } )'
```

Listing 2.11: SpatialWare geometry string for a triangle

2.5.4 MySQL Spatial

Open source database management systems like MySQL [MyS05] also support spatial data types and functions. Contrary to the other products, the spatial functionality of MySQL is directly built into the database system itself, namely in the MyISAM storage engine. MySQL's spatial functionality is a direct implementation of the OpenGIS Simple Features Specification for SQL (SFS) [OGC99]. Thus, it is closely related to the SQL/MM spatial standard. The complete spatial type hierarchy from the SFS is defined, with the inheritance rules and the majority of the functions and methods that operate on those types.

Some of the SFS functions are not available, most notably *ST_IsEmpty* and *ST_IsClosed* are missing. Also, MySQL does not yet support the management of different spatial reference systems. It is possible to attach a numeric identifier for an SRS with each geometry when it is constructed in the database, but that identifier has no effect whatsoever on the spatial operations. Most probably it is provided as a placeholder for future extensions. No coordinate transformations are implemented, implying that all spatial logic considers the geometries to be in the same two-dimensional data space. Therefore, no table or view to manage the spatial reference systems is necessary. In fact, MySQL omits all views of the spatial information schema. Especially no view is available to list all the spatial columns. The table COLUMNS of the MySQL system information schema has to be consulted instead.

Geometry values can be created in the database using the WKT and WKB representations only. Neither the ESRI shape format, nor the GML format, nor any other format are supported. The SFS did not introduce the handling of Z and M coordinates, and so MySQL does not implement any functionality targeted to that either. A set of non-standard functions tailored to the composition of the WKB format is provided. These functions take multiple parameters as input where each parameter is the WKB of a single parts of the final geometry. The functions return the WKB of the final geometry as result. For example, the function *LineString* takes an arbitrary number of points (as WKB) and constructs the WKB representation of the linestring that consists of the given points.

In order to overcome an inconvenience in the OpenGIS Simple Features Specification for SQL and the SQL/MM spatial standard, MySQL provides all functions with an abbreviated name also using a version with a long name. For example, the function *MPolyFromText* and *MultiPolygonFromText* are identical, where the second function follows a consistent naming convention and is intuitively easier to remember.

The spatial comparison functions in SFS are implemented. Beyond those, MySQL makes a set of functions available to compare the geometries solely based on their MBRs. For example, the function *MBRIntersects* takes two geometries as input, calculates their MBRs and then returns 1 (one) if the MBRs intersect and 0 (zero) otherwise.

2.5.5 Oracle Spatial

Oracle Corp. implemented an extension to their database system to handle spatial data. In fact, two related products are available. Oracle Locator is a light-weight feature of the Oracle 10*g* Standard Edition [Ora05c]. It comes with basic support for geometry types, indexing capabilities, and spatial routines. The Oracle Spatial product [Ora05f] builds on top of the Oracle Locator foundation. It defines additional spatial functionality and support for Z and M coordinates. The following description refers to Oracle Spatial.

Oracle Spatial defines a single type SDO_Geometry to represent spatial data in an Oracle database. The type hierarchy from the SQL/MM spatial standard is condensed into this one type, loosing strong typing. Different types of geometries are supported and the distinction between those is handled internally only (very much like MapInfo SpatialWare does it). The attribute SDO_GType of SDO_Geometry encodes the specific type of the geometry, along with some additional information. An application can extract the type through parsing that value and enforcing a specific type can be guaranteed with check constraints.

The standardized spatial functionality is fully present. Some deviations exist, for example Oracle Spatial uses consistently the prefix *SDO_* for the spatial routines instead of *ST_*. Furthermore, some spatial set operations carry their own names, most notably the operation to calculate the symmetric difference of two geometries is named *SDO_XOR* instead of the standardized *ST_SymDifference*. Geometry values can be constructed in an Oracle database by directly populating the attributes of the SDO_Geometry type. The WKT and WKB representations can be used as well since version 10*g*. These representations were not directly available in previous versions, hampering the interoperability then.

Like other products, Oracle Spatial offers the storing of Z and M coordinates in the geometries, along with operations that take advantage of that information. The so-called linear referencing feature is dedicated to the processing of M coordinates. For example, the function *Clip_Geom_Segment* can be used to extract the part of a geometry where the M coordinates (measures) are between the specified interval. Thus, the routine is the same as *ST_MeasureBetween* that is provided by the DB2 Spatial Extender.

We already discussed in Section 2.3.5 that the handling of spatial reference systems is rather simplified in the standard and that an approach like the EPSG geodetic parameter dataset [EPS04] should be adopted. Oracle Spatial already implements that, allowing a much more detailed control and management of SRS information. Offsets and scale factors to convert the floating point numbers for the coordinates are not used by Oracle Spatial. Instead, so-called tolerances are employed, and tolerances are associated with spatial columns or can be used directly as input parameters for spatial functions.

Geocoding as the process to convert an address to a geometry is supported. The geocoding is based on a set of tables stored in the database, and Oracle Spatial mandates how

the geocoding itself is to be done in terms of functions that need to be called. Beyond the handling of spatial vector data in Oracle Spatial, the additional products Oracle Spatial GeoRaster [Ora05d] and Oracle Spatial Topology [Ora05e] are available to support raster data and topological information, respectively.

2.5.6 PostgreSQL & PostGIS

The second open source spatial extension for a database system presented here is PostGIS [Ref05]. Refractions Research Inc. implemented it for PostgreSQL databases [PSQ05]. The implementation follows very closely the OpenGIS Simple Features Specification for SQL [OGC99]. Therefore, PostGIS also adheres to the SQL/MM spatial standard in a lose way.

PostGIS defines a single type named Geometry. This type is to be used to store geometric primitives like points, linestrings, polygons and homogeneous or heterogeneous collections of the primitives. Z and M coordinates can be kept in the geometry values, but as with all other products, no true three-dimensional operations are performed.

The lack of closure in the WKT and WKB representations with respect to the associated spatial reference system was addressed in PostGIS by prepending the numeric identifier of the SRS to the text or binary representation. The actual definition of the SRS has to be retrieved from the information schema, in particular the table SPATIAL_REF_SYS. The information schema itself consists only of the two tables GEOMETRY_COLUMNS and SPATIAL_REF_SYS as it was specified in the first version of the SQL/MM spatial standard.

An extension of PostGIS over the SFS and the SQL/MM spatial standard are functions that explicitly operate on the surface of a sphere or spheroid, e. g. *Distance_Spheroid* can calculate the geodetic distance between two points on the given spheroid. Another extension is the introduction of the function *AsSVG* that will convert the geometry as a SVG path data.

Part II

Functional Enhancements for SQL/MM

3 Spatial Application Development

The original intent of the development of SQL in the 1980s was to define a query language for end users to access data stored in a relational database. This idea has long been disbanded and today relational database systems are used by applications only [KBL05]. Software packages are layered on top of a database management system (DBMS) and communicate with it via well-defined and mostly standardized application programming interfaces (APIs), for example JDBC, CLI[1], or embedded SQL.

With spatial data (cf. Section 2.3.2) as first class citizens in a database, it is necessary to support spatial types in those APIs. But so far, the respective parts of the SQL standard do not specify any facilities tailored to spatial data. Neither does the SQL/MM spatial standard define such an infrastructure beyond the (insufficient) transform groups (cf. Section 3.2). Our final goal of this thesis is to represent geometries retrieved from a spatial database as objects in applications. It is also necessary to allow the geometry objects to be used directly as parameters in SQL statements. The classes for these objects shall come with the spatial functionality defined by the SQL/MM spatial standard. The current chapter explains how to extend the JDBC driver to accomplish this task.

Procedural and object-oriented programming languages like C [KR88], C++ [Str97] or Java [GJSB05] usually have an approach quite different from the relational model to manage data in an application. That is due to other programming constructs like arrays, pointers, and references being available in the programming language. This difference, also called *impedance mismatch*, requires a mapping between the relational world and the procedural world of applications. The interface for the mapping depends on the actual programming language. Spatial data with its structure that is inherently more complex than a simple integer value or a string increases the mismatch even further.

Java applications employ Java Database Connectivity (JDBC) [Sun01] or SQLJ for database connectivity. We introduce both interfaces in Section 3.1 while specifically focusing on the handling of spatial data (and more generally structured data types). Generally, the spatial values have to be transformed to a stream-based format. Transform groups as the only standardized mechanism to communicate spatial data between an application and a database system are described in detail in Section 3.2. Our specific focus laid on shortcomings of the transform functions. Accessing databases from Java applications is done through Java Database Connectivity (JDBC). We explain in Section 3.3 how to en-

[1] Call-Level Interface (CLI) is also known as Open Database Connectivity (ODBC). It was standardized by ISO in [ISO03j].

hance JDBC to return geometries as objects directly from result sets. Likewise, such geometry objects can be passed on to the DBMS via parameter markers without the need to go over an external representation. The chapter is closed with a summary in Section 3.4.

3.1 Spatial Support in JDBC

No class hierarchy compatible with the class hierarchy defined by Open Geospatial Consortium (OGC) and presented in Section 2.3.1 is part of the Java language specification. Similarly, the JDBC interface [Sun01] and the SQLJ [ISO03f] interfaces do not define any geometry class hierarchy. The SQL/MM spatial standard also does not extend those standards and specifications. This gap is even more surprising considering that the Open Geospatial Consortium defines a geometry class hierarchy along with associated spatial functionality [OGC05a]. That hierarchy is already implemented in several products, for example the GeoAPI [Geo05] or the Java Topology Suite [JTS03b].

The developers of JDBC already considered user-defined types in a relational database. Customized type mappings can be used when a value of a structured type or distinct type is fetched from a result set. The Java method *getObject* that is applicable for a *ResultSet* object takes the type mapping as additional input parameter. The type mapping is comprised of a *Class* object for each (fully qualified) data type in the database. The JDBC driver can then map the structured or distinct type to an instance of the specified class, which has to implement the *SQLData* interface. In particular, the methods *readSQL* and *writeSQL* of the interface play a vital role. The *readSQL* method is needed to decode the byte stream retrieved from the database and to initialize the Java object. Similarly, the *writeSQL* method performs the complementing operation and converts a Java object to the corresponding byte stream that can then be sent to the DBMS.

Customized type mappings can also work in conjunction with transform groups. Thus, a class implementing the *SQLData* interface has to decode (or encode) the data to the transformed format. In terms of the standardized spatial transform groups (cf. Section 3.2), well-known text (WKT), well-known binary (WKB), or GML representations have to be processed. Using transform functions, which requires an explicit setting of the specific transformation by the Java application, will cause the result set to indicate that the data type of the column is a BLOB or CLOB (depending on the actual transformation that took place) and not spatial type.

This technique is taking advantage of by the Informix Spatial DataBlade [IFX02b], which provides a type mapping for its spatial data types. *Listing 3.1* demonstrates how a point object can be retrieved from a result set with this Java package. It is assumed in the listing that a connection object named *conn* and a statement object named *stmt* already exist. The class *IfxSQLData* is part of the Java API of the Informix Spatial DataBlade, and its method *enableTypes* provides the mechanism to create the type mapping for the spatial data types. The query accesses a non-spatial column ID and the spatial column LOC from table T.

```
ResultSet rs = stmt.executeQuery("SELECT id, loc FROM t");
Map typeMap = IfxSQLData.enableTypes(conn);
while (rs.hasNext()) {
    int id = rs.getInt(1);
    IfxPoint pt = (IfxPoint)rs.getObject(2, typeMap);
    // process data ...
}
```

Listing 3.1: Extract point object from result set

The Java API of the Informix Spatial DataBlade uses the internal encoding of the spatial types in the database and decodes it directly to build the corresponding Java objects. It does not rely on any standardized external data format. This approach is quite suitable given the tight coupling to the DataBlade. A major drawback of this particular implementation is that it only allows the read access of geometries. The option to construct spatial Java objects and sending them to the DBMS as part of insert operations, for example, or binding them to parameter markers is not available. Additionally, the set of methods on the Java geometry classes is very limited. It includes only such functionality that can be easily computed in Java without requiring sophisticated spatial algorithms. For example, it is possible to count the number of points in a geometry, but the intersection of two *IfxGeometry* objects cannot be calculated. If such functionality is required, the application has to resort to the routines available in the spatial database or it has to implement them again as part of its own application logic. The second approach voids the integration of spatial functionality in an relational database management system (RDBMS) to a large part and again burdens the application with the spatial logic.

The final goal of a tight integration of spatial data into the Java language binding includes the availability of the full set of spatial routines at the application layer, while transparently exploiting the implementation of the existing spatial extensions in the relational database system. Additionally, the spatial class hierarchy as defined by OGC [OGC05b] should be included into the JDBC driver in such a way that a simple call to the method *getGeometry* retrieves a point or linestring object from a result set or an invocation of *setGeometry* binds a spatial object to a parameter marker.

Another part of the SQL standard is already concerned with the mapping between Java classes and structured types in the relational database. SQL/JRT [ISO03h] is intended for persistency frameworks that store Java objects in an RDBMS. An automatic mapping of classes to structured types and class members to attributes of structured types is defined by this standard. The structured types are created based on the Java class, and tables can be populated with the values of these structured types. Unfortunately, SQL/JRT is not applicable to SQL/MM spatial data types because the spatial extensions already exist and, therefore, a direct and automatic mapping between Java classes and the spatial types can usually not be established. The exact opposite direction of the mapping is needed.

3.2 Spatial Transform Groups

Transform groups are the standardized way to transfer values of structured types between a database engine and external code[2]. External code can either be an application or a user-defined routine that is executed in the context of an SQL session at a database server. The basic principle is that an external representation for a structured value exists where this representation uses non-structured data types only. For example, the external representations WKT or WKB are encoded in a character large object (CLOB) or a binary large object (BLOB), respectively. The WKT or WKB representation contain the same information as the corresponding values of type ST_Geometry.

A major advantage of transform groups is that the conversion of a spatial or, more generally structured, value does not have to be specified in each SQL statement by explicitly invoking the respective routine (e. g. *ST_AsBinary*). Instead, the spatial column can be used like any other expression in the select-list of a query. The database system will internally consult the transform group to determine which routine is to be called implicitly to convert the spatial values.

Multiple transform groups can be defined for each structured type. The SQL standard [ISO03i] specifies that the currently active transform groups are a properties of the current SQL session. The transform group is set for each type using the SQL statement `SET TRANSFORM GROUP FOR TYPE` *`type group`*. If no specific transform group has been selected for a certain type, then the default transform group is applied. This default can be modified with the SQL statement `SET CURRENT DEFAULT TRANSFORM GROUP` *`group`*. The selection of a transform group allows an application to process spatial values in different external representation without the need to modify the SQL statements to switch between the formats. SQL considers it as an error situation if no type-specific transform group was specified and no default group was set or if the one that is set does not apply for a type in question. It is unknown in this situation how the structured value is to be properly serialized.

The conformance of existing products to the SQL standard with respect to transform groups is sketchy at best. DB2 UDB provides transform functions as the SQL standard specifies. However, it is not possible to define the transform groups on a per-type basis [Sto05b] for a given SQL statement. Only a single transform group can be used and that one applies to all structured types that might be accessed in the statement. This restriction can quickly lead to problems in more complex scenarios where different structured types are involved. A single, common transform group covering all those types will be mandatory. Informix IDS does not use structured types but rather opaque types (cf. Section 6.1) to implement the spatial data types. Such values are treated like a byte stream in the interaction of an application with the spatial database system. Unless an

[2]External code comprises all components and modules that are not built into the database engine. Examples are applications and user-defined functions (UDFs).

explicit conversion functions is used, one has to decode the internal and undocumented encoding of the geometries in the byte stream. That is exploited by the Java API of the Informix Spatial DataBlade [IFX02b]. Oracle Database comes with a predefined transformation and serialization of its object types. Explicit conversions are necessary if another target format is required that deviates from the default serialization.

Subsequently, the three transform groups standardized in the SQL/MM spatial standard – *ST_WellKnownText*, *ST_WellKnownBinary*, and *ST_GML* – are introduced. The sample geometries shown in *Figure 3.1* are used to explain the formats. The formal syntax definitions for the formats are not given since they can be found in [ISO03d, ISO05b]. A single point p, a linestring l and a multi-polygon mp are used. The two polygons in the figure comprise a single logical unit and belong to the same geometry, the multi-polygon.

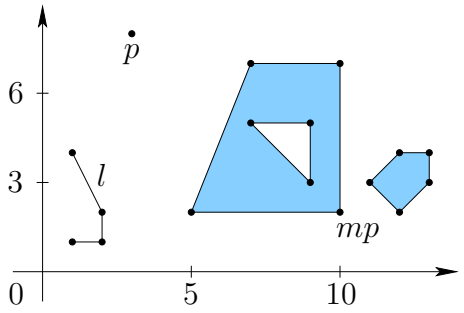


Figure 3.1: Sample geometries

3.2.1 ST_WellKnownText Transform Group

The *ST_WellKnownText* transform group converts a geometry to its well-known text (WKT) representation. WKT is a text based format that encodes the specific type of the geometry with a single keyword, followed by the coordinates for the single points. Coordinates of a single point are separated by spaces and commas distinguish between the points. Parentheses are used to group the structures in the geometry, e. g. points, rings, or single elements of a collection. Some examples based on the geometries in Figure 3.1 are given in *Listing 3.2*.

POINT (3 8)

LINESTRING (1 1, 2 1, 2 2, 1 4)

MULTIPOLYGON (((5 2, 10 2, 10 7, 7 7, 5 2), (7 5, 9 5, 9 3, 7 5)),
((13 3, 13 4, 12 4, 11 3, 12 2, 13 3)))

Listing 3.2: Sample geometries in WKT representation

WKT is easy to read for humans given its textual format. All textual representations are inadequate when coordinate information is involved. There are inevitable rounding issues with the conversion from the internal binary representation of a floating point number to its textual representation [Gol91]. Not all floating point numbers can be represented exactly with the same precision due to differing number systems.

3.2.2 ST_WellKnownBinary Transform Group

The only standardized binary representation for geometries is the well-known binary (WKB) format. Setting the transform group *ST_WellKnownBinary* for the spatial types in the SQL session implies that geometries are sent in the WKB format between the DBMS and the application. WKB encodes the specific geometry type with standardized numerical values, the type identifiers. *Table 3.1* aligns those identifiers to the spatial data types. The single-digit type identifiers were first assigned in the OpenGIS Simple Features Specification for SQL (SFS); the other identifiers were initially introduced in the SQL/MM spatial standard [ISO03d] to accommodate the additional data types in the type hierarchy. The structural information of the geometry is maintained in an array-like mechanism. The elements of such an array are stored sequentially and are preceded by the number of elements.

Spatial Type	Type Identifier	Spatial Type	Type Identifier
Point	1	MultiPoint	4
LineString	2	MultiCurve	1,000,004
CircularString	1,000,001	MultiLineString	5
CompoundCurve	1,000,002	MultiSurface	1,000,005
Polygon	3	MultiPolygon	6
CurvePolygon	1,000,003	GeometryCollection	7

Table 3.1: Numeric type identifiers in the WKB format

Listing 3.3 shows the hexadecimal representation of the WKB for the geometries in Figure 3.1. The output is formatted for a better visualization. The first byte (set in red) in each geometry indicates the byte order for multi-byte numbers, i. e. whether those values are encoded in little endian or big endian [Coh81]. The second through the fifth byte store the type identifier and are shown in blue. The number of points is needed for linestrings, multi-points or rings in a polygon. Golden color is used for this information in the listing. The eight bytes for each coordinate of the points are set in black. Thus, each point takes 16 bytes of space. Multi-polygons require additional data, namely the number of embedded polygons (set in purple) and the number of rings in each polygon (set in green). The embedded geometries of a collection are comprised of their own complete WKB, including byte ordering and type identifier.

```
x'0101000000000000000000084000000000000002040'

x'0102000000040000000000000000F03F000000000000F03F
  00000000000000400000000000000F03F0000000000000040000000000000040
  0000000000000F03F0000000000001040'

x'0106000000020000000
  0103000000020000000500000000000000000144000000000000040
  00000000000024400000000000000400000000000002440000000000001C40
  0000000000001C4000000000000001C400000000000001440000000000000040
  04000000000000000001C40000000000001440
  00000000000022400000000000001440000000000000224000000000000840
  0000000000001C400000000000001440
  0103000000010000000600000000000000002A400000000000000840
  0000000000002A4000000000000010400000000000002840000000000001040
  00000000000026400000000000000840000000000000284000000000000040
  0000000000002A400000000000000840'
```

Listing 3.3: Sample geometries in WKB representation

3.2.3 ST_GML Transform Group

The Geography Markup Language (GML) [ISO05b, OGC04] is another textual format besides WKT. The *ST_GML* transform group takes care of the implicit conversion of a spatial value in the database to its GML representation (or vice versa). The main difference to WKT is that the single elements are enclosed in XML tags and so the geometry structure is explicitly visible. The GML encoding of the geometries in Figure 3.1 is shown in *Listing 3.4*.

```
<gml:Point><gml:pos>3 8</gml:pos></gml:Point>

<gml:LineString>
   <gml:pos>1 1</gml:pos><gml:pos>2 1</gml:pos>
   <gml:pos>2 2</gml:pos><gml:pos>1 4</gml:pos>
<gml:LineString>

<gml:MultiPolygon>
    <gml:polygonMember>
        <gml:Polygon>
            <gml:exterior>
                <gml:pos>5 2</gml:pos><gml:pos>10 2</gml:pos>
                <gml:pos>10 7</gml:pos><gml:pos>7 7</gml:pos>
```

```
            </gml:exterior>
            <gml:interior>
                <gml:pos>7 5</gml:pos><gml:pos>9 5</gml:pos>
                <gml:pos>9 3</gml:pos>
            </gml:interior>
        </gml:Polygon>
    </gml:polygonMember>
    <gml:polygonMember>
        <gml:Polygon>
            <gml:exterior>
                <gml:pos>13 3</gml:pos><gml:pos>13 4</gml:pos>
                <gml:pos>12 4</gml:pos><gml:pos>11 3</gml:pos>
                <gml:pos>12 2</gml:pos>
            </gml:exterior>
        </gml:Polygon>
    </gml:polygonMember>
</gml:MultiPolygon>
```

Listing 3.4: Sample geometries in GML representation

GML is a very verbose way to encode a geometry as can be seen in the listing. A problem for the adoption of GML as interface between a spatial application and a database system can be found in its history. Several of the XML elements underwent significant changes in the course of the development. For example, there were various ways to encode coordinate information. In Version 2 of the specification, a point list could be set similar to the WKT representation by using a <gml:coordinates> element. The single points were separated by spaces and a comma is placed between the coordinates of a point. If a <gml:coord> element is used instead, an even finer structure applies to the XML document because each coordinate value is enclosed in another tag identifying the dimension, e. g. <gml:coord><gml:X>4</gml:X><gml:X>7</gml:X></gml:coord>. These two encodings are officially deprecated but still in use today. In fact, a list-based approach is still supported by GML, but the element is now named <gml:posList>. The semantics of commas and spaces are switched, i. e. a space separates the coordinates of a point and a comma separates the points. Depending on the underlying database system, the infrastructure to provide a native language binding option for spatial data will have to cope with all existing (and future) variations until the GML standard sufficiently stabilizes.

A second issue arises with the textual format. Same as for WKT, a conversion between the internal binary and the external GML encoding is required. Rounding issues are introduced, which are usually not desired at all. The different numbering systems are responsible for that because not all binary numbers can be represented exactly as decimal numbers and vice versa.

3.2.4 Transforms for Application Bindings

Choosing the proper transformation is highly dependent on the specific application. In general, the textual representations should only be considered if the spatial data is not to be processed any further, for example, if the data is included in an XML document or converted to Scalable Vector Graphics (SVG) [W3C03]. The mentioned rounding differences can be avoided with a binary format, e. g. *ST_WellKnownBinary*. The coordinates are handled as DOUBLE values according to the IEEE 754 standard [IEE85]. The coordinates are usually stored as such floating point values in the database, so a simple copying is sufficient.

The three standardized transform groups share the same problems with respect to transferring geometries between RDBMS and application. A geometry is not directly accessible as an object with associated functionality in the application. The application has to deal with a large object (CLOB or BLOB) and take care of the necessary conversions to other data structures. Such data structures have to be defined by the application itself. The same applies to additional transform groups that products may have implemented. For example, the DB2 Spatial Extender provides a transform group named *ST_Shape* to transfer geometries in the ESRI shape format [ESR98].

Once the external representation of a geometry is retrieved from the database system, it must be parsed by the application and a spatial object constructed for further processing. The reverse direction requires the construction of the external representation by encoding it in a large object and sending that to the database system. Both conversion tasks are not integrated into the language binding mechanisms like JDBC or CLI. Furthermore, spatial functionality, for example the computation of the intersection of two geometries, is not natively available to applications and has to be reimplemented in the application layer. Only packages independent of any RDBMS and JDBC, e. g. JTS [JTS03b], are available. Summarized, integration of the spatial data into the layer responsible for the data exchange between the RDBMS and the application is by no means a seamless solution today.

3.3 Enhancements to JDBC

Java and the JDBC language binding offer an ideal base line to support spatial data. Based on the work in [Sal06b], a spatial class hierarchy is integrated natively to the JDBC driver, exceeding the available techniques of the customized type mappings (cf. Section 3.1). It is the responsibility of the JDBC driver to create a spatial object when a corresponding geometry is retrieved from the database. The *ResultSet* interface is enhanced to include the respective getter method, i. e. *getGeometry*. Additionally, spatial objects can be bound to parameter markers in SQL statements using applicable setter methods (e. g. *setGeometry*) for the *PreparedStatement* interface. The third affected area

concerns stored procedures for which the class *CallableStatement* provides an interface. Stored procedures may combine input and output parameters and those parameters could also be geometries.

3.3.1 Spatial Class Hierarchy

The spatial class hierarchy that Salzbrenner included into the JDBC driver adheres very closely to the OGC class hierarchy specified in [ISO04a]. *Figure 3.2* depicts the hierarchy using the notation of the Unified Modeling Language (UML) [BRJ05]. The names of abstract (non-instantiable) classes are set in italics. The class hierarchy incorporates some adjustments already made for the SQL type hierarchy by omitting the two classes *Line* and *LinearRing*. Objects of such classes are to be represented using *LineString*. The second modification refers to empty geometries. In Section 2.3.8, we already discussed the shortcomings of the implicit handling of empty geometries by intermixing it into all other types or classes. Therefore, the *Empty* class is added. Further adjustments to the spatial class hierarchy – like the realignment of geometry collections and geometric primitives as was done in Figure 2.10 for the spatial SQL type hierarchy – are not necessary. Java is a procedural language and the *composite* design pattern [GHJV95] can be directly exploited with the available language features without overly complicating the required logic in applications.

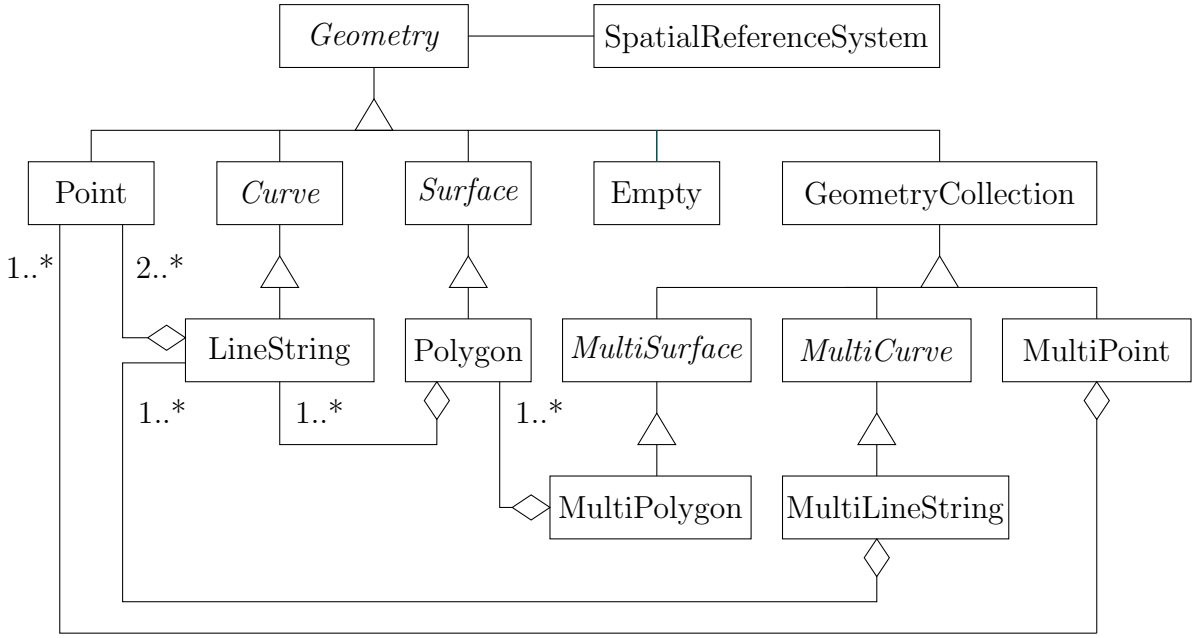


Figure 3.2: Spatial class hierarchy for the JDBC driver

The methods for the spatial classes are essentially a direct mapping from the spatial routines that are already available in a spatial database. Methods are provided for all four functional groups:

- convert between geometries and external data formats,
- retrieve properties or measures from a geometry,
- compare two geometries with respect to their spatial relation, and
- generate new geometries from others.

Methods exist to compare two spatial objects whether they spatially intersect or to calculate the length of a linestring, for example. Additionally, the Java classes provide further methods to manipulate geometries, e. g. adding elements to geometry collections, adding and deleting rings in polygons, modifying points in linestrings and polygons, concatenating two linestrings etc. That way, spatial objects can be constructed incrementally. The traditional constructors based on external data formats like WKT, WKB, and GML are provided as well.

The various methods can either be implemented natively in Java (as JTS already does, for example) or the functionality existing in the spatial database can be used when needed. For the second approach, it is necessary to transfer the spatial object to the DBMS, construct the geometry value there, invoke the matching method and then transfer the result back to the application. If the result is again a geometry, the corresponding spatial object will be implicitly created. *Listing 3.5* shows the SQL statement that can be used by the Java class *Geometry* in the method *getIntersection*. Both geometries to be intersected are converted to their WKB representation *wkb1* and *wkb2*, respectively.

```
VALUES ST_Geometry(wkb1).ST_Intersection(ST_Geometry(wkb2))
```

Listing 3.5: Falling back to the DBMS for spatial calculations

The implementation effort for the spatial support in JDBC can be significantly reduced by passing calls to spatial method through to the spatial database system. This approach has an impact on the performance of spatial methods, but the full spatial power can be leveraged right away. The prototypical implementation in Section 3.3.5 applies this technique demonstrating the overhead implied by the additional communication and conversion logic.

We recommend to add the JDBC spatial class hierarchy to the `java.sql` package because the classes and interfaces that need to be extended for spatial are placed in this package. With the addition of the spatial class hierarchy it is necessary to add the type information to the class *java.sql.Types*. This class defines constants that are used to identify generic SQL types. A native support for spatial data requires an identifier `GEOMETRY`. The more specific classes also need identifiers and `POINT`, `LINESTRING`, etc. are added.

3.3.2 Extending the Interface ResultSet

Result sets are a client-side representation of results of an SQL query. A cursor concept is implemented to provide access to the rows of a result set and a series of so-called getter methods, for example *getString*, is available to access the single attributes of the row on which the cursor is currently positioned.

With the advent of native spatial data support in JDBC it becomes necessary to handle spatial data types like any other data type, for example INTEGER. To that end, a method *getGeometry* is added to the interface *ResultSet*. Contrary to [Sal06b], no further methods are needed. It is not necessary to add methods like *getPoint* or *getLineString*. The task to down-cast a geometry to a more specific class can be accomplished with a regular Java-style cast as in *Listing 3.6*. The listing is based on the precondition that a *Statement* object named `stmt` was already created and is available. The Java Virtual Machine will raise a runtime exception in case of an error if the geometry class and the target class of the cast are not compatible, e. g. if an attempt is made to cast a point geometry to a *LineString* object.

```
ResultSet rs = stmt.executeQuery("SELECT id, loc FROM t");
while (rs.hasNext()) {
    int id = rs.getInt(1);
    Point pt = (Point)rs.getGeometry(2);
    // process point data ...
}
```

Listing 3.6: Fetching a point from a *ResultSet*

3.3.3 Extending the Interface PreparedStatement

Prepared statements offer a way to parameterize SQL statements; therefore, they can be executed several times with variations for the specified parameters. The DBMS prepares the statement once, e. g. it is compiled and then cached along with its execution plan. Thus, multiple compilations of similar statements can be avoided and overall performance is usually improved.

The parameters in such a prepared SQL statement are indicated by a question mark '?' and represent SQL expressions. Before the SQL statement can actually be executed, a value has to be bound to each parameter marker to complete the statement.

One example to exploit prepared statements in the spatial world is to use a query that retrieves all geometries from a certain table where the geometries intersect with a given, parameterized geometry. Window queries are very typical for that. Native support for spatial objects in JDBC mandates that spatial objects can be directly bound to parameter markers as in *Listing 3.7*. The listing assumes that a *Connection* object

named conn is available in the current scope of execution. The cast expression in the SQL statement is necessary because a single parameter marker is untyped and the DBMS could not successfully prepare and compile the statement.

```
PreparedStatement prepStmt = conn.prepareStatement(
        "SELECT id, loc FROM t " +
        "WHERE  loc.ST_Intersects(" +
        "          CAST(? AS ST_Geometry)) = 1");
// bind geometry (rectangle) to parameter marker
Geometry g = new Polygon(
        "POLYGON(10 10, 10 20, 20 20, 20 10, 10 10)");
prepStmt.setGeometry(1, g);
// execute prepared statement
ResultSet rs = prepStmt.executeQuery();
while (rs.hasNext()) {
    // fetch and process each row's values ...
}
```

Listing 3.7: Binding a point to a *PreparedStatement*

Listing 3.7 uses the method *setGeometry* to bind a geometry to a parameter marker. This method is added to the class *PreparedStatement* specifically for spatial objects. Two overloaded versions of this method are needed with the following signatures:

1. *setGeometry*(int *parameterIndex*, Geometry *geom*)
2. *setGeometry*(int *parameterIndex*, Geometry *geom*, int *targetType*)

The first version is to be used when the given geometry shall be interpreted as value of type ST_Geometry in the database and not one of the more specific spatial data types. The second method provides additional type information – based on the class *java.sql.Types* – so that the methods for the more specific data types can be used in the SQL statement. As an example, *Listing 3.8* demonstrates how the SQL method *ST_X* is invoked by an SQL statement. This method is only defined for the type ST_Point, which makes it mandatory that the JDBC driver knows that a point geometry is provided to the Java method *setGeometry*. The second parameter is responsible for that. JDBC can then call the proper constructor of the specific subtype.

```
PreparedStatement prepStmt = conn.prepareStatement(
        "VALUES CAST(? AS ST_Point).ST_X()");
// associate geometry (point) with parameter marker
Geometry g = new Point(10, 20);
prepStmt.setGeometry(1, g, java.sql.Types.POINT);
// execute prepared statement
ResultSet rs = prepStmt.executeQuery();
```

Listing 3.8: Using specific types for parameter markers

3.3.4 Extending the Interface CallableStatement

Stored procedures can have three types of parameters: `IN`, `OUT`, and `INOUT`. `IN` parameters are used to pass parameters to the procedure, whereas `OUT` parameters are set by the procedure and the value is returned to the caller. `INOUT` is a combination of both where the caller provides a value but the same parameter also returns a value. Additionally, a procedure may return result sets by opening cursors and leaving the cursors open beyond the end of the procedure execution.

If any of the result sets includes spatial columns, the handling of those does not require any special considerations. Such result sets are treated like result sets returned from the Java method *Statement.executeQuery*. We already described the mechanism to retrieve geometries from result sets in Section 3.3.2 where the method *getGeometry* was defined for the interface *ResultSet*. Thus, result sets from stored procedures are automatically covered for the handling of spatial data.

Only the parameters of the procedure remain to be taken care of. `IN` parameters do not require any enhancements to the interface *CallableStatement*. The necessary *setGeometry* methods are already inherited from the interface *PreparedStatement*. `OUT` (and also `INOUT`) parameters make the addition of a method *getGeometry* to the class *CallableStatement* mandatory. This method is defined with the same parameters and semantics as for the interface *ResultSet*. *Listing 3.9* illustrates how a stored procedure can be invoked with the explained additions to return a point geometry via an `OUT` parameter. As usual, the base line for the code snippet is that a JDBC connection object named `conn` is already available.

```
CallableStatement callStmt = conn.prepareCall(
        "{CALL spatial_proc(?)}");
// register parameter marker as OUT parameter
callStmt.registerOutParameter(1, java.sql.Types.POINT);
// execute procedure and retrieve parameter
callStmt.execute();
Point p = (Point)callStmt.getGeometry(1);
```

Listing 3.9: Retrieving spatial objects from stored procedure `OUT` parameters

Another aspect where callable statements are affected by the spatial data types comes with the *registerOutParameter* that was also used in Listing 3.9. This method informs the JDBC driver that the procedure returns a value for the indicated parameter position. At the same time, it makes the expected return type known. The signature of the method does not have to be modified because the spatial types identifiers are added to the class *java.sql.Types* and, thus, fit in right away. However, the internal implementation needs to be aware of the spatial objects.

3.3.5 Prototypical Implementation

We implement the concepts for the native support of spatial objects in JDBC and the associated necessary enhancements prototypically. The prototype is based on DB2 UDB Version 8.2 and its type 4 JDBC driver. The implementation covers the spatial class hierarchy as in Figure 3.2 and the new methods to the Java interfaces *ResultSet* and *PreparedStatement*. Additionally, the spatial classes provide a full set of spatial methods to make spatial functionality directly available to the application.

Initially, [Sal06b] implemented a completely new spatial class hierarchy in the JDBC driver. This hierarchy was oriented towards the modified and improved spatial type hierarchy presented in Section 2.3.8. However, the majority of the spatial functions was not directly coded in Java. Instead, all computations were deferred to the spatial extensions in the RDBMS. *Figure 3.3* illustrates the control flow. An invocation of the method *getNumPoints* (on a *Geometry* object) in the application causes the geometry WKB to be generated. The geometry object initiates the execution of a SQL statement at the DBMS to construct an ST_Geometry value and invoke its associated *ST_NumPoints* method. The result is then passed back to the application via the *Geometry* object.

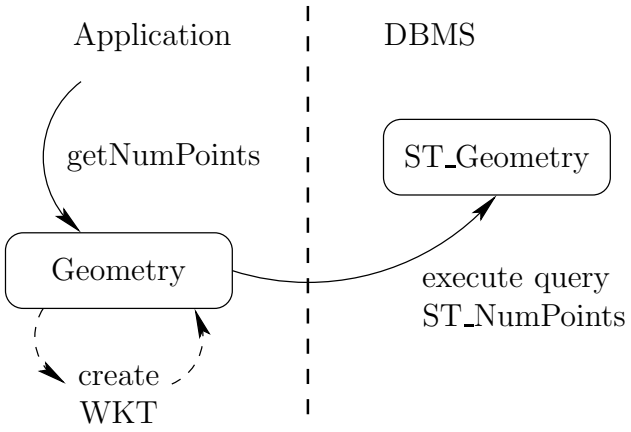


Figure 3.3: Referring to DBMS for spatial computations

Although that approach required less implementation effort, it turned out to be impractical for studying purposes. No comparisons between a native Java implementation and the deferred execution in the spatial database could be made. Therefore, the spatial Java class hierarchy in the JDBC driver was replaced with the Java Topology Suite (JTS) [JTS03a]. JTS comes with a class hierarchy that is compliant with [OGC99]. The switch to JTS deviates from the modified SQL type hierarchy, but that did not turn out as an obstacle. The procedural approach of Java already prefers a traversal of elements in an array as is being used by geometry collections to implement the *composite* design pattern [GHJV95]. The JTS classes already include many spatial methods, e. g. *equals* that compares two geometries for spatial equality.

Unfortunately, JTS is not (yet) integrated into JDBC. We closed this gap and adjusted each spatial method to support both execution models. Depending on an environment setting, JTS either executes the spatial logic itself or it defers to the DB2 Spatial Extender. Thus, a comparison between both approaches can be made.

Using WKB for the Spatial Data Transfer

DB2 UDB does not fully support type-specific transform groups (cf. Section 3.2). We had to work around this issue in the prototype. That is done by resorting to the explicit conversion routine *ST_AsBinary* to retrieve geometries in their WKB representation. WKB has been chosen to minimize rounding issues compared to the textual formats. The ESRI shape format [ESR98] as the other alternative is also a binary format but that was discarded because it does not retain the full type information. It does not distinguish between linestrings and multi-linestrings or polygons and multi-polygons. For example it is not obvious whether a linestring is a geometric primitive or a multi-linestring that is comprised of a single element only.

The call to *ST_AsBinary* is added transparently to the SQL statements executed via the Java method *executeQuery* as *Listing 3.10* illustrates for the spatial column LOC in the given query. The original SQL *query* provided by the caller is nested as subselect in `SELECT` statement. The final `ORDER BY ORDER OF` clause is a DB2 extension to SQL that can be used to retain the ordering of indicated subquery for the rows in the final result.

```
SELECT ..., loc..ST_AsBinary()
FROM   ( query ) AS temp
ORDER BY ORDER OF temp
```

Listing 3.10: Introducing the *ST_AsBinary* function call

The reverse way to pass spatial objects as parameters to an SQL statement always implies the use of the factory function *ST_Geometry*. To that end, we have to modify the SQL statement itself, and a basic parser is responsible for this task. Due to the complex nature of SQL, the parser is very limited and only supports simple SQL statements. Either a spatial method has to be applied directly to the parameter marker, or the marker is to be used in an `INSERT` or `UPDATE` statement in a place where the column type can readily be detected, i. e. the values-clause or the set-clause, respectively.

Implementing the Enhancements

It is not a viable option to modify the source code of the IBM DB2 JDBC driver itself. The source code is not freely available. However, the number of viable alternatives is rather limited since enterprise applications are rarely built on open source database

systems like MySQL. The development also revealed that the closed source nature of the JDBC code did not impose an insurmountable obstacle and that a simple work-around could be used to implement the new functionality.

With the DB2 JDBC driver, it would have been possible to extend the interfaces *ResultSet* and *PreparedStatement*, but the DB2-specific classes that implement those interfaces cannot be extended. Also, rewriting just those DB2 classes that implement *ResultSet* and *PreparedStatement* is not doable. *ResultSet* objects are created by *Statement* objects and such objects are instantiated from a *Connection* object in turn. These two classes would have to be reimplemented as well, which implies that a major portion of the driver would have to be touched.

The approach chosen by [Sal06b] is more light-weight in that only a simple wrapper around *ResultSet* is coded. The interface for result sets is extended by deriving a *SpatialResultSet* from it and adding the *getGeometry* method there. The class *SpatialDB2ResultSet* implements this interface. The code of the class focuses only of the new method. All other methods are simply passed through to the result set object that was created by DB2 and is nested in the *SpatialDB2ResultSet* object. *Figure 3.4* depicts this architecture. The arrows indicate the control and data flow between the various objects at the different levels.

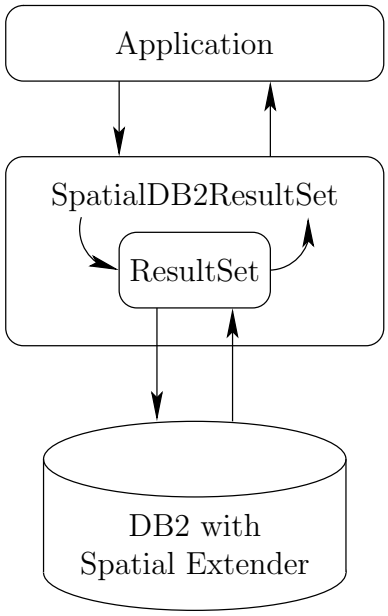


Figure 3.4: Wrapping regular *ResultSet* objects

JTS's spatial classes have a straight-forward internal implementation. A *Point* is comprised of two coordinates, one for the X and one for the Y dimension. Instances of *LineString* use an array of coordinates. JTS also implements a class *LinearRing*, which

is derived from *LineString* and imposes the condition that the first and last point in the array have to be identical, i. e. that the linestring is closed. Linear rings are used for the definition of polygons. *Polygon* objects always carry an outer ring. Additionally, an array is present to contain the rings for any holes, the inner rings. Finally, a *GeometryCollection* consists of an array of *Geometry* objects. The derived classes of *GeometryCollection* only constrain the types of objects that can be stored in the array, but the structure itself is not different.

The OpenGIS Simple Feature Specification mandates the implementation of various methods for the spatial classes, and JTS conforms to that. All geometries provide methods like *isSimple*, *isEmpty*, or *getLength* to query properties. The relationship between two geometries can be tested with a multitude of predicates, e. g. *touches* and *intersects*. The third group of functions allows to derive new geometries from existing ones and all methods that are also implemented in the SQL/MM spatial standard are available, for example *buffer*, *getCentroid*, and *intersection*. A shortcoming is, however, that only the WKT representation is supported as external format by the method *toText*. Neither WKB nor GML are implemented. Given that the geometries are to be transferred as WKB between JTS and the database system, a WKB reader and writer had to be added to accommodate for this lack of functionality.

A set of selected methods are modified in JTS to provide a mechanism to compare the native JTS implementation of spatial functionality with the deferral to DB2. To that end, the environment variable `JTS_DEFERRED_EXECUTION` is consulted. Depending on its setting, the native implementation or the DB2 Spatial Extender is selected. In the latter case, an SQL statement is constructed and sent to DB2 as we outlined in Section 3.3.1.

Performance Results

The option to rely on the DB2 Spatial Extender for spatial calculations comes with a performance penalty due to the communication of the geometry data between the JDBC driver and the database system. On the other hand, the DB2 Spatial Extender is highly optimized for spatial operations and – as is demonstrated below – the communication overhead can be compensated for non-trivial spatial operations. A ThinkPad T30 powered by a 2 GHz Intel processor and equipped with 1 GB of physical main memory was used for the performance measurements. The Linux kernel release 2.6.13 was used as installation base for DB2 UDB Version 8.2.5 (i. e. FixPak 12).

The shapes of the countries of Europe were used as test data. *Table 3.2* lists a selected subset of the geometries with their number of points and number of polygons in each geometry along with the name of the represented country. The number of points is an indicator for the complexity of the geometry and the size of its WKB representation.

Figure 3.5 visualizes the performance for a single call to the Java method *getNumPoints* that corresponds to the DB2 Spatial Extender routine *ST_NumPoints*. The fig-

Country	Number of points	Number of polygons
Monaco	13	1
Switzerland	1275	1
Portugal	2954	35
Germany	6162	50
Ireland	9012	94
Finland	15710	653

Table 3.2: Sample data for performance measurements

ure demonstrates that there is a constantly growing gap between both implementation variants. The time difference can be nearly completely attributed to the communication overhead to convert the Java spatial object to its WKB representation, passing this WKB to DB2 and creating the corresponding ST_Geometry value.

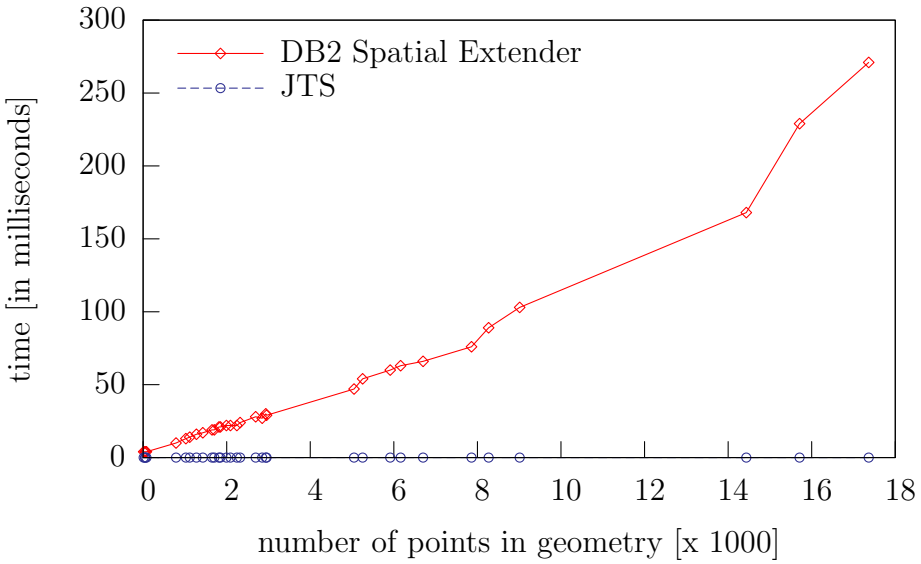


Figure 3.5: Performance of *getNumPoints* method

The number of points is directly accessible in an attribute of the SQL geometry value so that no further computations (beyond the construction) are necessary. The Java implementation for the *getNumPoints* method iterates over the various JTS-internal arrays and sums the number of points. That can be done very quickly and is always completed in less then a microsecond on the test data.

It is a very simple operation to determine the number of points of a geometry. Therefore, difference in the execution times for the method *getNumPoints* in Figure 3.5 are not surprising. The situation is quite different if true spatial calculations are performed. *Figure 3.6* illustrates the execution times for the Java method *equals*. The first curve

shows the time needed to transform the two geometries being compared, sending both to DB2 and invoking the routine *ST_Equals* of the DB2 Spatial Extender. Using the JTS method to evaluate the spatial predicate resulted in the second curve.

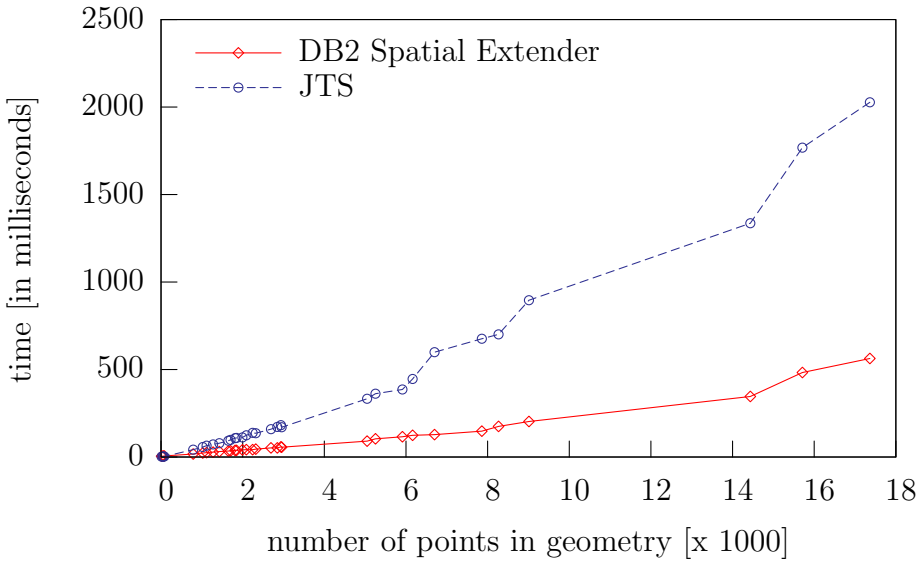


Figure 3.6: Performance of *equals* method

The DB2 Spatial Extender out-performs JTS by a factor between 2.5 and 4.5, except for very small geometries with approximately 100 points or less. With more complex geometries, the evaluation of the spatial predicate in JTS is comparably slow. Java as a an interpreted language (after being byte-compiled) cannot compete with the compiled C/C++ code of the DB2 Spatial Extender. Even the overhead added by the communication of the spatial values to the database system is more than compensated. It makes the approach to resort to the RDBMS worthwhile for complex spatial functionality.

3.3.6 Conclusions for the Extended JDBC Driver

Native support for spatial data in the JDBC interface is doable and it comes with a significantly improved usability compared to transform functions and customized type mappings. The methods to retrieve spatial objects directly from the database or to assign spatial objects to parameter markers require only minimal intrusions to JDBC. At the same time, the full range of spatial functionality can be supported either with a pure Java implementation (provided by JTS) or by deferring to the DB2 Spatial Extender.

The prototypical implementation proved that the underlying spatial extension can be exploited to provide the full range of spatial functionality to Java applications. An application does not need to be aware that the spatial database system is consulted to

come up with the result. However, a noticeable performance penalty comes with this approach because additional transformation and communication overhead is incurred. Implementing the spatial functionality in Java code will not be as optimized as compiled C/C++ code. Thus, the overhead may become irrelevant for expensive spatial operations.

An automatic decision between a local evaluation and the referral to the spatial database system can be directly implemented in the methods of the spatial class hierarchy in JDBC. The complexity of all involved geometries can be analyzed, for example by counting the number of points. The complexity in combination of other factors like network throughput can tip the decision in the desired way.

The concepts and results can be directly applied to the SQLJ standard [ISO03f]. Implementations of SQLJ are built on top of JDBC. SQLJ even allows the derivation of a JDBC *ResultSet* object from an iterator. Thus, the method *getGeometry* will be available as well. Similarly, prepared and callable statements can be used in SQLJ.

Given that JDBC is not an international standard but instead is rather closely guarded by Sun Microsystems, it may not be possibly to actually extend the JDBC specification [Sun01] itself. Similarly, describing the necessary extensions in the SQL/MM spatial standard [ISO03d] is not a viable option as it would interfere with copyrights and it would introduce a dependency of the standard on external entities. However, the SQLJ standard is developed by ISO. Thus, the SQL/MM spatial standard can and should include the definitions to add the native support for spatial data. That is done by identifying the respective clauses in SQLJ itself and unambiguously phrasing the necessary modifications.

3.4 Summary

Neither the SQL standard nor the SQL/MM spatial standard provide a deep integration for structured types, especially spatial data types, in the language binding mechanisms that are used by application to communicate with a relational database system. The problem of the impedance mismatch is even further increased with spatial data. Currently, the standards resort to transform groups to serialize structured values so that they can be sent as character or byte streams to an application, which is then responsible to parse the stream and to create any corresponding objects.

The native support for spatial objects in JDBC can be established easily. The Java interfaces *ResultSet*, *PreparedStatement*, and *CallableStatement* have to be considered since those are the only places where values are communicated between the application and the DBMS. We added the methods *getGeometry* and *setGeometry* together with respective type identifiers in the class *java.sql.Types*. That is sufficient to cover all requirements.

Other object-oriented programming languages besides Java can be supported in a very similar way. The concepts described for JDBC can be directly adopted. For example, PHP [Hud05] and Perl [WCO00] exploit the CLI and ODBC interfaces. Those language could use their own structures to model geometry data. The techniques described for JDBC can be encapsulated in the programming language, providing effectively the same results functionality-wise. The spatial computation can be performed via SQL statements in the database system by explicitly converting the spatial values to or from their WKB representation and invoking a routine of the spatial extension. That mechanism offers a quick solution to at least provide the functionality even if it does not come with the best performance expectations.

4 Integration of Graph Functionality

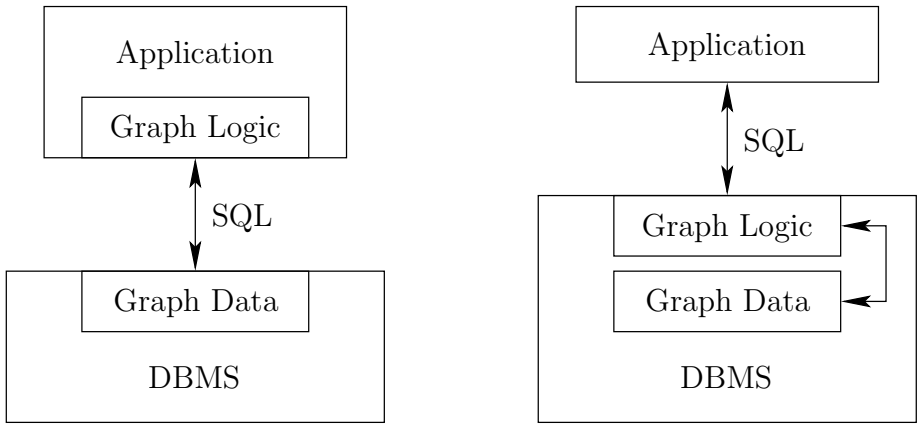
The spatial functionality standardized in the SQL/MM spatial standard [ISO03d] is primarily concerned with modeling geometric objects in relational database systems. Contrary to geometries, graphs are concerned with the connectivity between objects, i. e. topological properties. Graphs and geometries are tightly related, however [ZRS02]. For example, a geometry that represents a parcel in a land registry database has inherent topological properties because it is adjacent to other geometries (parcels). Likewise, a street network is often stored as a set of linestrings in a spatial database. A direct relationship between such a network and a graph is obvious. Calculating the shortest path between any two points in a network defined by a set of linestrings is not an easy feat to accomplish. Employing graph techniques for the task is much more straightforward given that a graph is a much more adequate data structure and that research has laid a strong foundation in graph theory [Die05].

Many applications of relational database systems can take advantage of graphs for their necessary processing. Graph management with the aid of relational database systems is very important in a large number of areas, for example geographic information systems (GISs) where routing operations are quite common [BM98], biological tasks like genome mapping [GBL95], and various other areas such as software reengineering [LWS01]. Trees and hierarchies are special graphs, and their use is very common in virtually any field of application [Cel05]. In short, the integration of graphs into the relational database management system (RDBMS) can be worthwhile if graphs and set-oriented processing of SQL are to be combined.

For spatial data, the integration of graph functionality becomes obvious when it is considered that real-world objects can usually move only on a predefined set of trajectories as specified by the underlying network, e. g. roads, railways, or rivers. Thus, the important measure is the distance between two points in the network and not only the Euclidian distance that is supported by the SQL/MM spatial standard.

Previous work on the integration of graphs in (spatial) relational databases is rather scarce [PZMT03]. Today's database management systems generally do not support native functions to process and analyze graphs. Because of that limitation, applications only use a database to store the graph data but implement the functionality for analyzing graphs in the application layer. For example, a GIS that deals with transportation networks stores the underlying network as geometry information in the database. The topological information may be stored in the database or it may be derived by the

application when needed. In any case, the graph-specific logic like shortest path computation, determination of the maximum flow within the network, or intelligent traffic management needs to have the graph data structure available at the application layer. *Figure 4.1(a)* depicts this situation. Managing graphs directly inside the database system as in Figure 4.1(b) may avoid performance issues resulting from communication of graph data between the application and the database management system (DBMS) when navigating through a graph. Additionally, it makes the graph functionality available to a broader user base at the same time.



(a) In application (b) Integrated in DBMS

Figure 4.1: Graph processing for database-oriented applications

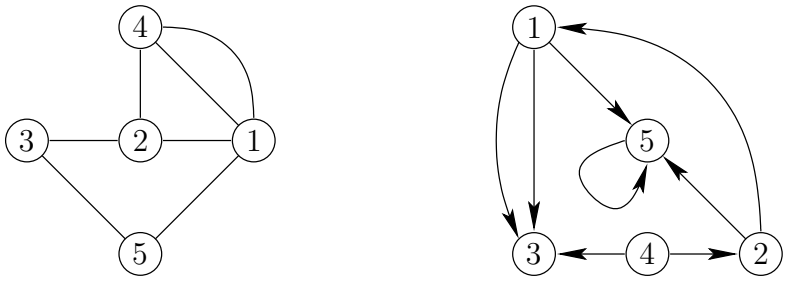
We dedicate the current chapter to the development of techniques specifically for integrating graphs with spatial data. The base line for the storage of spatial data is provided by any of the various products presented in Section 2.5, i. e. products that conform to the SQL/MM spatial standard. Our declared goal is that graph operations can be applied to spatial columns and the underlying graph acts as an indexing mechanisms and, thus, remains hidden from the application. First, we present the concepts and definitions in Section 4.1. Section 4.2 gives an overview on existing work related to the handling of graphs in (not necessarily relational) database contexts. So far the research was only focused on graphs in non-relational spatial databases. We close this gap by defining forward and reverse mappings between geometries and graphs in Section 4.3. The different types of geometries are analyzed separately. These concepts were implemented in a Spatial Graph Extender on top of DB2 UDB. We explain the details of this implementation and its usage in SQL statements in Section 4.4. Additionally, results proving that the performance of the extender is acceptable are presented. A full integration of graph technology in the kernel of a database engine imposes additional requirements. Section 4.5 goes into the details and explains how those requirements can be met. Finally, we close the chapter with a summary in Section 4.6.

4.1 Graph Concepts

The formal definition of graphs are explained in this section. The most common algorithms operating on graphs are introduced. In particular, shortest path calculations play an important role in real-world applications.

4.1.1 Definitions

A graph G is a pair (V, E) where V is a non-empty set of vertices in the graph, and E is a binary relation on V representing the edges of the graph [Die05]. Graphs can be distinguished in directed and undirected graphs. The elements of the set of edges E are ordered pairs (u, v) with $u, v \in V$ for directed graphs (also known as *digraphs*) where each edge has a defined direction. Such an order is not necessary for undirected graphs and, therefore, each element of E is a set $\{u, v\}$ with $u, v \in V$. An undirected graph can also be modeled using the facilities for directed graphs simply by duplicating each edge and reverting its direction, i. e. the direction doesn't matter any longer if each edge (a, b) has a complementing edge (b, a). *Figure 4.2* shows an example of an undirected graph $G_1 = (V_1, E_1)$ where $V_1 = \{1, 2, 3, 4, 5\}$ and $E_1 = \{\{1, 2\}, \{1, 4\}, \{1, 4\}, \{1, 5\}, \{2, 3\}, \{2, 4\}, \{3, 5\}\}$ and a directed graph $G_2 = (V_2, E_2)$ where $V_2 = \{1, 2, 3, 4, 5\}$ and $E_2 = \{(1, 3), (1, 3), (1, 5), (2, 1), (2, 5), (4, 2), (4, 3), (5, 5)\}$.



(a) Undirected (b) Directed

Figure 4.2: Examples of graphs

By enhancing the tuple (V, E) to a triple (V, E, f) where f is a function that assigns a real number w to each edge $e \in E$ a graph becomes a *weighted* graph. According to that, w is also called the *weight* of edge e.

Two more terms are important for graphs: *incident* and *adjacent*. An edge $e_1 = \{u, v\} \in E$ where $u, v \in V$ of a graph $G = (V, E)$ is incident to the vertices u and v. That means, an edge is incident to the vertices it connects. Two vertices $n, m \in V$ are adjacent if and only if there exists an edge $e_2 = \{n, m\} \in E$ (or $e_2 = (n, m) \in E$ for digraphs) in the graph.

A *path* between two vertices $u, w \in V$ of a graph $G = (V, E)$ is a sequence of vertices $\langle v_0, \ldots, v_k \rangle$ where $u = v_0$, $w = v_k$, and $(v_{i-1}, v_i) \in E$ with $i = 1, 2, \ldots, k$. Thereby, k determines the length of the path. Accordingly, a vertex w is called *reachable* from a vertex u if there exists a path between u and w.

The *neighborhood* of a vertex $v \in V$ of a graph $G = (V, E)$ is a set of vertices that are adjacent to v. A formal definition of the neighborhood $N(v)$ is $N(v) = \{u \in V | \{u, v\} \in E\}$. Each vertex has a *degree* which defines the number of adjacent vertices. The degree of a vertex v is defined by $deg(v) = |N(v)|$.

General graph theory also distinguishes between finite and infinite graphs. Infinite graphs consist of an infinite number of vertices. For all practical purposes, graphs managed in database systems are always finite. They have a finite number of vertices and also a finite number of edges.

4.1.2 Algorithms

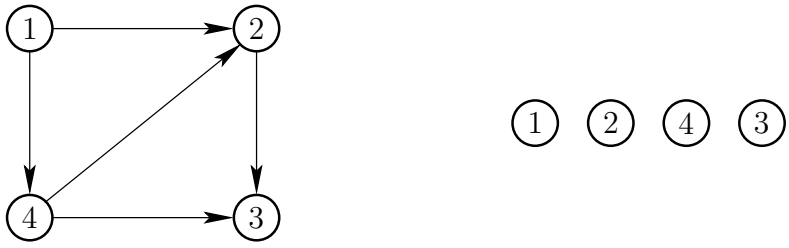
A multitude of different algorithms that work on graphs are known in the literature [Sed88]. We briefly describe four of the more well-known algorithms below as they have a wide field of application purposes in graphs and trees as specialized graphs. Special considerations are given to the *shortest path* algorithm because the current working draft of the SQL/MM spatial standard [ISO05a] already includes the *ST_ShortestPath* function. This function has a very similar purpose for linestring data as the shortest path algorithm has on regular graphs.

We only focus on search and graph traversal algorithms because those play an important role for operations that exploit graphs in SQL queries as in Section 4.4.5. Additionally, search algorithms are a good help to understand graph concepts. Graph modifications, i. e. additions and deletions of vertices and edges are discussed and analyzed in the literature [CLR01].

Breadth-First Search

The breadth-first search is one of the simplest graph algorithms. It can be used to traverse all vertices of a graph in a particular order, starting at a given vertex, the *start vertex* s. The order of the traversal is determined by the length of the path each vertex has from s: the shorter the path, the earlier the vertex is visited. Note that each vertex is visited only once via its shortest path; and a vertex is only visited if there actually exists a path from s to that vertex.

Figure 4.3 shows an example of a small graph and a resulting list of nodes visited during the breadth-first search if the start vertex was chosen to be 1 (one). The result does not have any preference between vertices 2 and 4 that have both the same path length from the start vertex.



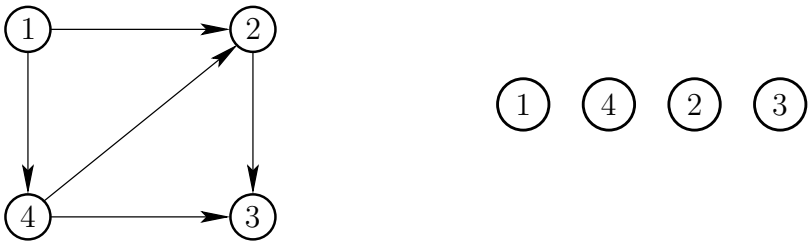
(a) Graph (b) Breadth-first traversal

Figure 4.3: Example for breadth-first search

Depth-First Search

The depth-first search is – similar to the breadth-first search – a traversal algorithm. The major difference is that one path beginning at the start vertex s is followed as far as possible and all the vertices on that path are listed in the output in their sequence on the path. Once the end of the path is reached and no further adjacent and not yet visited vertices can be found, the algorithm tracks back to the next to last vertex and tries to find another not-yet-visited adjacent vertex there. This backtracking is performed until all vertices of the graph or all paths beginning at s are visited.

An example for a depth-first search is shown in *Figure 4.4*. The start vertex is 1 (one). Next, the algorithm follows the path to vertex 4 and on to 3 where this path ends since there is no outgoing edge from the last vertex. It tracks back to vertex 4 and finds a path to vertex 2 and the complete graph is traversed once the algorithm reaches there. The final tree that can be constructed from this particular traversal is also shown in the figure. Note that there is no preference whether the first step traverses to vertex 4 or 2. No order is implied by the vertices so that both ways are possible.



(a) Graph (b) Depth-first traversal

Figure 4.4: Example for depth-first search

Minimum Spanning Tree

A spanning tree is a (sub)graph T that connects all the vertices of the given graph G by using only the edges of G. Spanning trees can be generated, for example, using the breadth-first search or depth-first search algorithms introduced above. Vertices can exist in a graph that are not reachable from certain other vertices. Thus, not every graph may have a spanning tree.

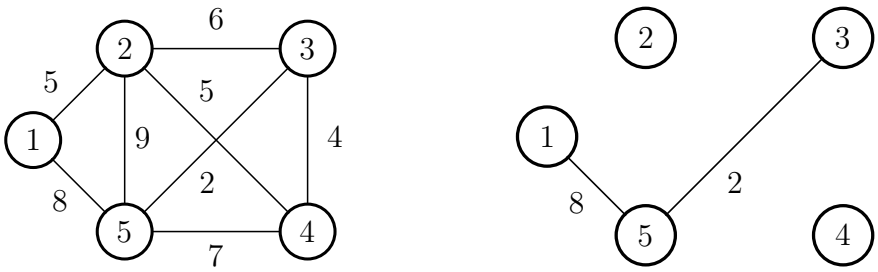
A minimum spanning tree (MST) is a special case of a spanning tree where the weights of the graph edges are used to rank the different spanning trees. The MST is the spanning tree where the sum of its edges' weights is minimal in comparison to all other spanning trees for the same graph. Various algorithms are proposed in the literature to compute an MST, for example the algorithms developed by Kruskal and Prim [CLR01, Sed88]. An example of a minimum spanning tree is shown in *Figure 4.5*. Note that it does not matter which start vertex was chosen in the case of undirected graphs. The MST for directed graphs might well be different, however, depending on the start vertex.

(a) Graph (b) Minimum spanning tree

Figure 4.5: Example for minimum spanning tree

Shortest Path

A very common and important operation on a graph $G = (V, E)$ is the calculation of the shortest path from one vertex $v \in V$ to another vertex $w \in V$. The shortest path is only along the edges $e_i \in E$ and the sum of the edges' weights is minimal. Many different algorithms are known in graph theory to calculate the shortest path or a semi-optimal shortest path, for example Dijkstra's shortest path algorithm [Dij59] or the A* algorithm [Nil80]. A simple example shown in *Figure 4.6* demonstrates the shortest path from vertex 1 to vertex 3. As one can see, several other paths from 1 to 3 exists, for example 1-5-4-3, 1-2-3, or 1-2-4-3, but only the path 1-5-3 comes with the minimal sum of edge weights.



(a) Graph (b) Shortest path from 1 to 3

Figure 4.6: Example for shortest path

4.2 Literature & Product Overview

Graph indexing (i. e. using graphs as access paths) for topology information stored in relations in an RDBMS is discussed in [ZZ94]. Zhao's basic assumption is that the road map database already stores the vertices and edges of the graph explicitly in node and link tables. A graph index is constructed from that data and subsequently maintained to keep it synchronized with the data in the tables. Spatial vector data as the source for the graph is not considered. Therefore, the mandatory forward and reverse mappings between geometries and graphs are not available.

The combination of spatial and graph-related queries was already analyzed by [HJR97]. The spatial queries include overlap operations whereas the graph-related portions are concerned with finding shortest paths, for example. Shortest path queries use varying weight criteria like finding the shortest path distance-wise or including the average speed on streets in a street network and, thus, come up with the fastest (as opposed to the shortest) route. Four different strategies for processing such combined queries were discussed by Huang. However, the mapping between geometries and graphs is not considered. Additionally, a file system based storage mechanism was employed instead of a relational database system. An integration of the graph-specific logic with already existing relational data (including existing spatial data) is not possible.

A formal algebra for graph operations in spatial databases is presented in [EG94]. The specifications of the algebra are completely independent of spatial and relational database systems and their specific constraints. The operations are defined in a formal syntax. Thus, the results from the paper are only partially applicable in the subsequent sections. In particular, no beneficial information for the indexing of spatial vector data using graphs or networks is provided.

Another approach to supply graph functionality in a relational database system is described in [Güt94b]. SQL is extended with the so-called `DERIVE` statement to formulate queries on graphs. The graphs are managed and used as explicit, first class objects (like tables and functions) in the database. Extending the query language is not an option for

users who want to take advantage of graph technology today. Therefore, the currently available facilities of the relational database systems must suffice, which is contrary to the approach in the paper. Besides, spatial data in relational databases is not mentioned, and the representation of geometries in graphs is not discussed either. Similar extensions of SQL are proposed by [MS90] and [CMW87], requiring a modification of the database engine.

[GSVGM98] focuses on information retrieval and keyword searches using proximity metrics. The database is viewed as a graph with data values represented in vertices and relationships between datums form the edges. A distance function is employed to establish the relationships. Objects mapped to vertices may be records, single values, or even parts of a value. This approach is not directly applicable to build graphs from relationally managed spatial data as Section 4.3 outlines because no RDBMS is in the picture. Besides, the mapping between geometries and graphs is completely left open. One could interpret the points of a linestring as vertices in a graph and the Euclidian distance may be the means to define the relationships.

Directed graphs are employed for BANKS [ABC+02] to model tuples in a relational database as vertices. Referential integrity constraints (foreign keys) define the edges in the graph. The goal of BANKS is to make an RDBMS accessible for unstructured keyword searches. It does not handle spatial data and does not implement routing-based operations on spatial vector data in SQL.

Even farther from the goal to integrate graph functionality in existing relational database systems is the model described in [GPBG94]. There, all data is represented in a graph-based object-oriented fashion. Not only the schema is inherently a graph, but also the data manipulation employs a language expressed by graph transformations. This concept is contrary to the relational model and SQL as a query language that we adopted as a given base line ourselves.

Oracle Corp. is the only database vendor so far that considered the need for graph support and provides the Topology and Network Data Models package [Ora05e]. This package allows the user to manage topological information in dedicated (relational) tables in an Oracle database. Topology can comprise way more information beyond graph structures. For example, enclosed areas may be managed together with their relationship to vertices, edges and even other, neighboring areas. The package provides operations like finding the nearest neighbors or the shortest path in the network (or graph). A network can be constructed from geometries; a reverse mapping from the network to the geometries is not available, however. For example, a network may be derived from linestrings representing streets. However, an application accessing the database has to develop its own means to get the linestrings of the streets that actually participate in the shortest path from a starting point to a target point in the network. Even if this functionality appears to be closely related to the *ST_ShortestPath* function in the latest working draft of the SQL/MM spatial standard [ISO05a], it falls short of the standardized logic.

4.3 Mappings Between Geometries and Graphs

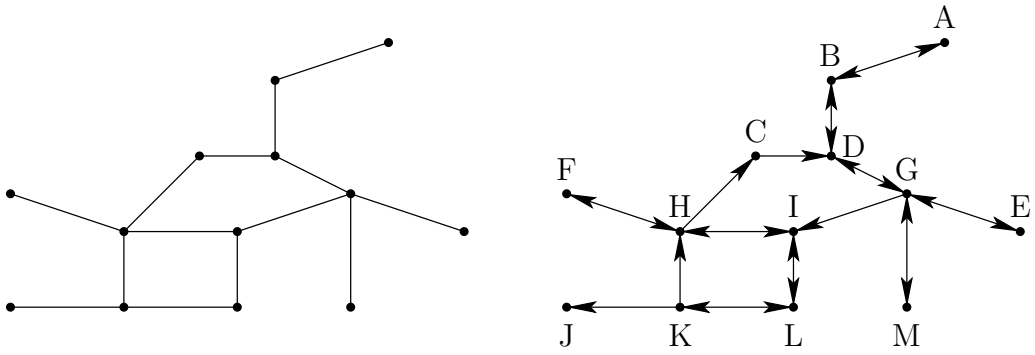
The integration of graph functionality into an object-relational DBMS is a building block for integrating graph functionality with spatial vector data processing. Besides the straight-forward mapping from linestrings to graphs, this section goes into two other schemes to transform points and polygons into graphs such that graph operations can be applied on the geometry data. We cover not only the forward mapping but also explain how a reverse mapping may be achieved. The graph functionality can be used in SQL statements for the spatial data. Thus, the functionality is directly related to the current working draft of the SQL/MM spatial standard [ISO05a], even if it covers a much broader range and allows the straight-forward integration of further graph algorithms.

Oracle Topology and Network Data Models [Ora05e] is already available as a product dedicated to graph functionality. It comes with a forward mapping to construct graphs from linestrings. Graphs are not treated as internal structures similar to indexes; there is no reverse mapping to retrieve the original linestrings as results of a graph functionality. The Spatial Graph Extender that we present in Section 4.4 is targeted at closing these gaps.

Graphs are to be used as indexing structures for spatial data. That means, graph operations are used as filters for geometries, e. g. to find only those geometries that form the shortest path between two points. To that end, it is necessary to define a mapping between geometries in a spatial table and graph structures. The basic concepts for the mapping are similar to the mechanisms described in [OSQZ02] where topological information is transformed into geometries and vice versa. The idea for the conversion is that vertices and edges in the graph directly relate to points and line segments in a geometry, respectively. If all the parts belonging to a geometry can be identified in the graph, the geometry (or its relevant part) can be extracted from the graph. Some sort of identifier has to be managed in the graph for this purpose. Given that the graph is actually built from geometries, this identifier is already available at construction time. It merely has to be retrieved from the database and stored in the graph at the appropriate places, e. g. in the adjacency lists.

4.3.1 Building Graphs from Linestrings

The most intuitive mapping of spatial data to graphs applies to linestrings or collections of linestrings. If a table in a spatial database contains all streets of a certain region, then a graph can be constructed from those streets where intersections become vertices of the graph and the streets themselves are mapped to edges. The resulting graph has to be directed since one-way streets may exist in the real world and the direction is relevant for routing applications. *Figure 4.7* illustrates such a mapping. The points in Figure 4.7(a) represent the points that are used to define the linestrings. Each of those points becomes a vertex in the graph.



(a) Linestrings (b) Graph (directed)

Figure 4.7: Mapping linestrings to graphs

An analysis of the sample graph in Figure 4.7(b) shows that there are points that solely connect two line segments. For example, the vertices B, C, and L have a degree of two. They are derived from points marking a bend in the street and not an intersection of two streets. The vertices for such points are not essential for the graph and can be removed without loosing any information relevant for algorithms operating on the graph. Thus, the graph can be simplified to the one shown in *Figure 4.8*.

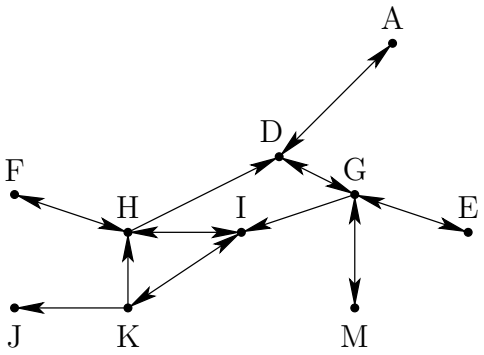


Figure 4.8: Simplified graph with non-essential vertices removed

In real-world applications, the linestrings describing the streets are more detailed, implying that each linestring contains many such points that are not needed in the final graph. For example, the shapefiles with the street information for the 50 states of the United States of America show that 60 to 80%, sometimes even 90% of all the points in the spatial data would be vertices with a degree of two (cf. Section 4.4.7). Removing those non-essential vertices from the graph leads to a tremendous reduction of the graph size and, thus, to a significant speed-up of the graph algorithms.

There are several considerations applicable on the way how the graph is constructed and reduced, i. e. the removal of non-essential vertices. Those situations are considered below. Finally, we specify the complete mapping between linestrings and graphs.

Edge Weights

Naturally, a graph derived from linestrings must be a weighted graph, for example, to retain length information. When a vertex is removed, the edges incident to the vertex are removed and replaced with a new edge that directly connects the vertices incident to the removed edges. The weights of the removed edges are retained by adding them and using the result as the weight for the new edge as *Figure 4.9* demonstrates. If, for example, the weight stands for the length of street segments, the new edge carries the correct, combined length. The numbers at the edges in the figure indicate the respective weights. The direction of the edges is omitted because these (and the following) considerations apply to directed and undirected graphs equally.

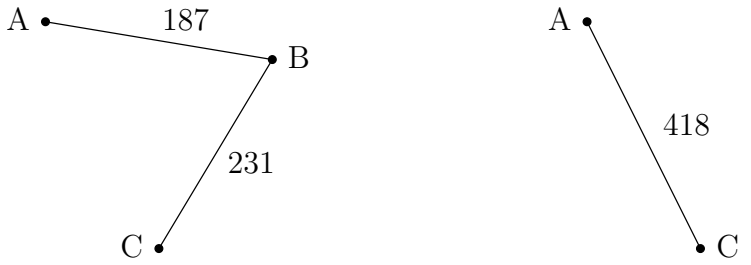


Figure 4.9: Retaining weight of edges incident to non-essential vertices

Maintaining Identifiers of Linestrings

A graph constructed from linestrings needs to maintain the information from which linestring a certain edge in the graph is derived. That is necessary to enable the reverse mapping to relate parts of the graph back to the tuples in the relational table from which the graph was derived. In order to avoid the excessive storage of the complete linestring in the graph and to use the spatial comparison function *ST_Equals*, it is more efficient to store the identifier (primary key) for the tuple along with each edge. Thus, an edge carries its weight and additionally the key value (identifier).

The key value has a major task in reducing the graph. Considering the example where a street may change its name along the way without an intersection being present. These are two streets in the database and one street has a point as end point and another uses the very same point as start point. Mapping this point to a vertex in the graph results in a vertex with a degree of two. This vertex would qualify as a non-essential vertex that could be removed from the graph. However, routing operations like shortest path queries may be used to generate a verbal description of the route. That usually includes the information about the name of the street. It is important to know when the name of a street changes and the vertex in the graph is needed to have this event available. The key value for each edge has to be used to exclude such vertices from the

list of non-essential vertices, i. e. treat it as essential. *Figure 4.10* illustrates how vertices (B, C, E, and F) incident to two edges with the same key value can be removed. The vertex (D) connects two different linestring. Therefore, it must be retained during the graph reduction.

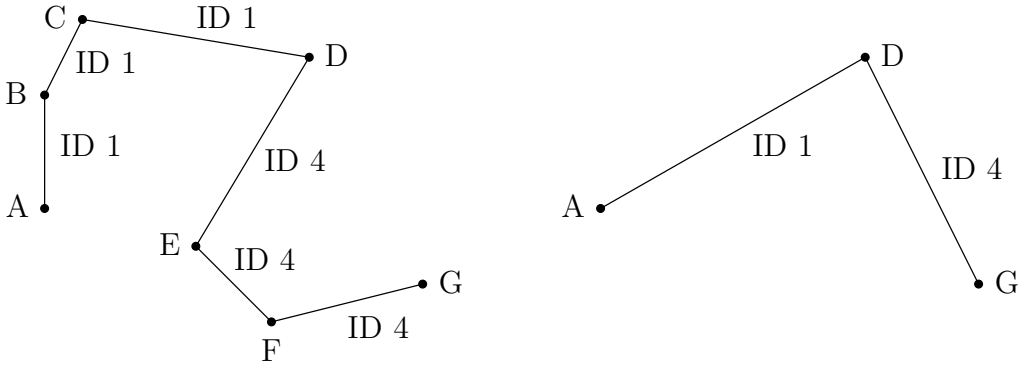


Figure 4.10: Retaining vertices connecting two different linestrings

Disconnected Cycles

Real-world street networks may contain cycles that are represented by single linestrings. For example, a round-about like the one in *Figure 4.11(a)* might exist without a connection to any other street. The successive reductions of the graph will lead to the situation in Figure 4.11(d) with only a single vertex remaining in the cycle. This vertex would be removed as well since it is considered as non-essential. In the end, the complete linestring would vanish from the graph.

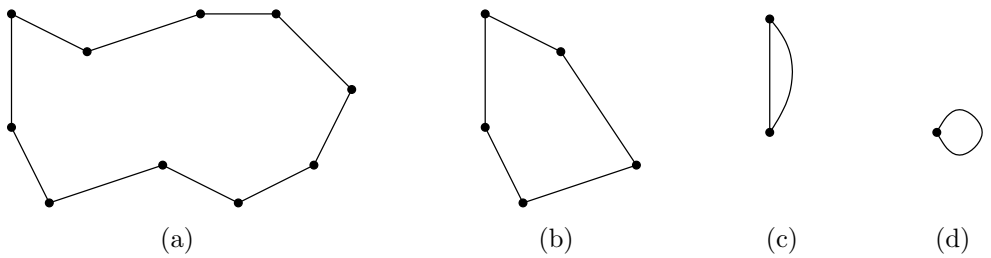


Figure 4.11: Disconnected cycles

It is necessary to keep such cycles in the graph to avoid any loss of data in the graph. Thus, the cycle can be reduced to a single vertex as in Figure 4.11(d) but single vertices connected to themselves must always be essential.

Duplicated Linestring Segments

Another particularity of street networks is that such networks contain multiple linestrings that have common segments as in *Figure 4.12*. Sometimes, two streets are even completely identical. Such scenarios can happen if a single street carries two names and is listed under both names in the spatial database. For example, the California state highway number 9 in Santa Clara County also has the name "Big Basin Way". The common segments can have several common points, leading to shared vertices. Reducing the graph by testing vertices for a degree of two misses these vertices as their degree is four (or larger).

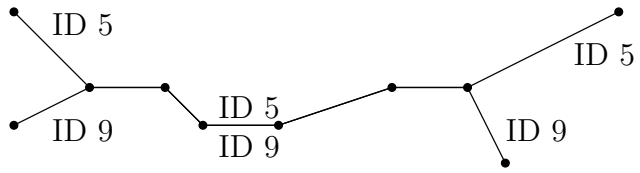


Figure 4.12: Multiple linestrings with common segments

Although such duplicated segments could be ignored, it leads again to an unnecessary increase of the graph size. Therefore, the concept of non-essential vertices is extended to include all vertices that have one or more pairs of incident edges and both edges in a pair carry the same key value. Additionally, all pairs connect to the same two adjacent vertices.

Storing Coordinates in Vertices

All vertices in a graph must have a unique identifier. No artificial identifier like an integer number can be used because vertices are derived from points that originally define linestrings. Thus, a point p has to map to a vertex v for all linestrings that contain p along their way. Therefore, the identifiers of all vertices shall be the points themselves. The representation of the points may be adjusted, if necessary.

The coordinates available in the vertices deriver another benefit besides the implicit mapping of the same points from different linestrings to the same vertex. The reverse mapping can be supported directly. Finding the shortest path in a graph usually does not imply that the path always starts at the beginning at a linestring (street) and ends at the last point. Instead, only a segment of a linestring might be part of the shortest path. The result of the shortest path operation includes the exact coordinates of the points/vertices where the traversal of a particular linestring begins and where it ends. It is now easily possible to create a point geometry from that start vertex or end vertex and use the geometry in spatial functions and methods. For example, it allows for the exact rendering and visualization of the shortest path. In particular, an application itself does not have to perform any further processing involving the linestrings on the shortest path to find the necessary intersections.

Forward Mapping

With all the special situations considered, the mapping between linestrings and graphs can be completely specified. The graph includes only the vertices for the essential points according to Definition 2.

Definition 2 (Essential point)
A point is essential in a set of linestrings if and only if any of the following conditions applies to its corresponding vertex in the graph:

1. *the vertex is incident to at least two edges derived from different linestrings,*
2. *the vertex has not exactly two adjacent vertices, or*
3. *the vertex is not incident to pairs of incoming and outgoing edges only, where the elements of each pair originate from the same linestring.*

The essential points are those that cannot be omitted from the graph without loosing information. The non-essential points can be excluded from the graph. Thus, the mapping between linestrings and graphs adheres to the following rules:

1. Each essential point of a linestring is translated to a vertex in the graph. The point identifies the vertex uniquely.
2. Each part of a linestring that connects two essential points becomes an edge.

Only the points in the definitions of linestrings result in vertices in the graph. The spatial intersection of two linestrings does not lead to a vertex in the graph if there is no common point in both linestrings at the intersection. *Figure 4.13* illustrates this case where the intersection is marked with the circle but no explicit point exists in either linestrings at this location.

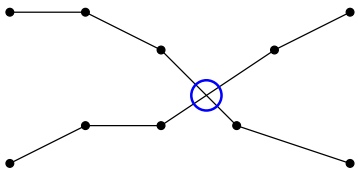


Figure 4.13: Intersections of linestrings without common point

The approach not to consider intersections if the two linestrings do not share a common explicit (inner) point is contrary to the implementation in [Ora05e]. Such situations occur frequently in real-world scenarios. For example, a street may cross another over a bridge but there is no intersection of both streets. Highway crossings are typical for that and additional ramps establish a connection between both streets. Routing operations must consider this situation and handle it correctly.

Reverse Mapping

Graph algorithms like shortest path queries return a subgraph, i. e. a part of the graph. When such algorithms are used on graphs derived from a set of linestrings, then the result should not be a subgraph but rather a list of linestrings (or sections of linestrings) that describe the shortest path. Thus, a reverse mapping from the graph to the linestrings from which the graph was constructed is necessary.

The basic element for the reverse mapping is the key value stored for each edge in the graph. The linestrings from which the edges originate can be identified with those key values, together with the information about the table where the linestrings are stored. The graph algorithms return the vertices and edges as their result. Thus, the key values are inherently available as well. The graph algorithm returns a table that lists for each edge its incident start vertex u and end vertex v along with the weight and the key value. For each edge, the following steps are processed:

1. Query the table T for which the graph was built to find the linestring l from which the edge was derived. The key value is used for that since it uniquely identifies l in T.

2. The vertices u and v encode the coordinates of the respective points. The point geometries s and e are constructed, respectively. If the following edge in the result set represents the same linestring, then the next end vertex v can be used instead to construct e.

3. A spatial method *ST_ExtractSegment* operating on L and taking s and e as input is used to cut the segment of the linestring between both points. This segment is then returned.

The method *ST_ExtractSegment* is not defined in the SQL/MM spatial standard. It should be added for the ST_LineString and ST_MultiLineString data types because its functionality is mandatory for the reverse mapping.

The reverse mapping may return a table where each row represents a segment on a single linestring. Alternatively, the graph-related routines could return just a single linestring (or multi-linestring) by aggregating all linestrings into a single geometry. Today's products do provide such aggregation functionality, for example the function *SE_Dissolve* of the Informix Spatial DataBlade does just that. Even if the underlying DBMS does not provide the infrastructure for user-defined aggregates, work-arounds can be used [SC02]. However, aggregation is generally not preferable since aggregation looses all information associated with single segments. In particular, it is not possible to join the table returned by the graph operation with the original table (for which the graph was built). Additional data like street names cannot be matched to segments in the shortest path, for example.

The above explanations assume that a graph is constructed for a spatial table in the relational database system. The function *ST_ShortestPath* to be introduced in the next version of the SQL/MM spatial standard takes a multi-set of rows as input, together with the start and end point for the shortest path calculation. The function does not relate the shortest path calculation to existing tables. That differentiation is not an issue for us because the multi-set is essentially a temporary table. The graph can be built, the shortest path calculated, and the result mapped back to the linestrings in the temporary table as was described before. Consequentially, the functionality described here can be aligned with the SQL/MM spatial standard.

The *ST_ShortestPath* function in the current working draft of the standard is highly inappropriate for routing operations. First, it is mandated that the start and end points for the calculation have to be a start or end point of a linestring. Next, the function does not return a single linestring or several linestrings for the shortest path but rather an array of key values. Each key value identifies a linestring participating in the shortest path. The value refers to the original data. The specific segment of the linestring is not known, leaving it to the callers of the function to detect the relevant portion themselves. Finding the relevant portions is particularly hard to do if the shortest path travels along a segment on a linestring, leaves this linestring and then comes back to the same linestring at a later point as in *Figure 4.14* where the shortest path (framed with the dotted line) between s and e is calculated. The key values for the two linestrings are shown. The two dashed segments are both returned as key value 1 (one). Both of these problems do not exist with the reverse mapping specified above.

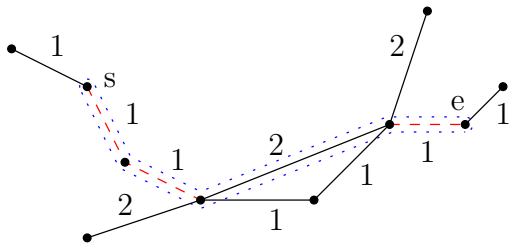


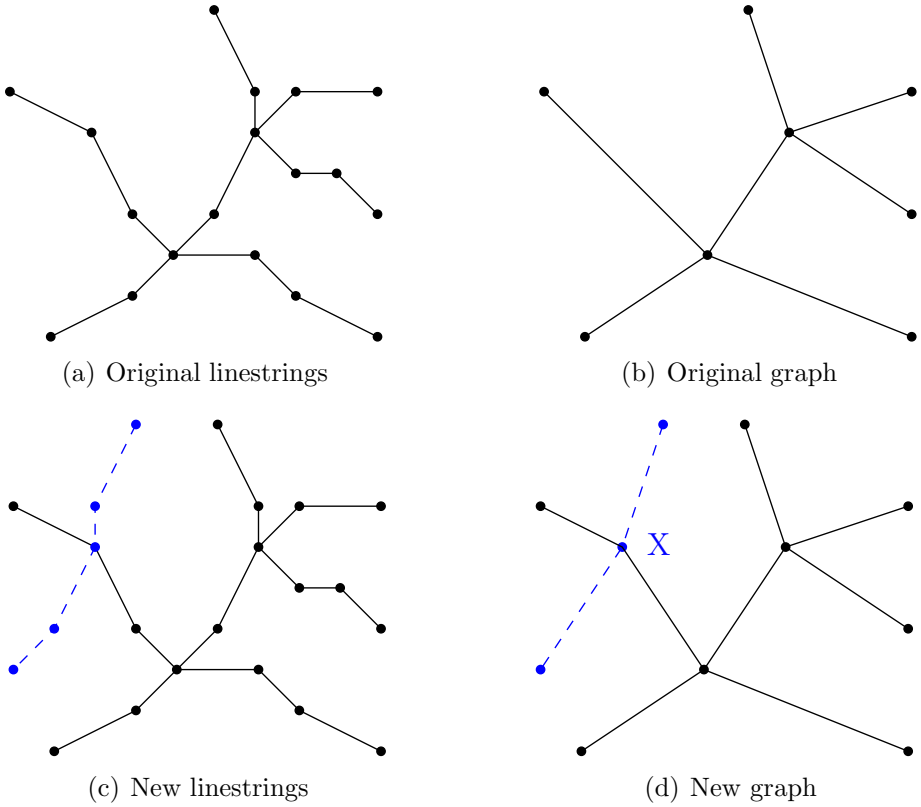
Figure 4.14: Non-identifiable shortest path from *ST_ShortestPath*

Graph Updates

The mapping introduced so far assumes that a static set of linestrings is used to build the graph. However, spatial data may change over time. New linestrings are added or existing ones are deleted or modified.

When new linestrings are inserted into the spatial relational table, the graph has to be adjusted to include all essential vertices and edges for the new linestring. An issue arises now because non-essential vertices cannot be determined solely by the existing (reduced) graph and the new linestring. The new linestring might change a non-essential vertex to

an essential one. For example, the situation in *Figure 4.15* shows some linestrings and the (reduced) graph built from them (Figure 4.15(b)). The insertion of a new linestring (shown as dashed line in Figure 4.15(c)) leads to the graph in Figure 4.15(d). The essential vertex X in the graph stems from the new linestring, but only in conjunction with another linestring from the original graph.



(a) Original linestrings (b) Original graph

(c) New linestrings (d) New graph

Figure 4.15: Inserting new linestrings

It is necessary to perform additional work to correctly determine essential and non-essential vertices in the process. First, all linestrings in the table that intersect with the new linestring are extracted and vertices are created in the graph for all points of the linestring – essential and non-essential ones. Thus, the graph is reconstructed completely in the affected area. The vertex X in the figure will exist in the graph. Then, the new linestring is mapped to the graph by adding all its points as vertices and the creation of the respective edges. At this stage, all newly created and reconstructed vertices are analyzed whether they are essential or not. The vertex X will now be considered as essential because the edges incident to it originate from two different linestrings. Non-essential vertices are removed as a final step, leading to the graph in Figure 4.15(d). The described logic is summarized in *Algorithm 1*. Pseudo code is used to clearly describe the single steps.

Algorithm 1 Inserting new linestring

```
1: NV ← ∅
2: nl ← new linestring to be mapped to graph
   /* reconstruct non-essential vertices in affected area */
3: for all linestrings l that intersect nl do
4:    remove edges originally derived from l using key of l
5:    for all points p of l do
6:       map p to vertex v and add v to graph
7:       NV ← NV ∪ {v}
8:    end for
9:    add edges for l to graph
10: end for
   /* insert new linestring */
11: for all points p of nl do
12:    map p to vertex v and add v to graph
13:    NV ← NV ∪ {v}
14: end for
15: add edges for nl to graph
   /* remove non-essential vertices */
16: for all v ∈ NV do
17:    if v is non-essential then
18:       remove v from graph
19:    end if
20: end for
```

Alternatively, the points at the intersections of the new linestring with the previously existing linestrings may be extracted with spatial operations. However, the spatial routine *ST_Intersection* cannot be used to find those points because the linestrings might not have a common explicit point at the intersection (cf. Figure 4.13) – but the routine would return such a point as its result. Therefore, the intersecting linestrings have to be found and then their definition must be parsed to find the actual points. This operation is also simpler than the computation of the intersection, but it requires another method to be added to the SQL/MM spatial standard for this specific purpose.

Yet another approach would be to rebuild the complete graph from scratch. This is inefficient due to high construction time for the complete graph. The partial reconstruction of the affected subgraph is very much preferable. We demonstrate in Section 4.4.7 that the construction of a graph can require several minutes, depending on the underlying spatial data.

A deletion of a linestring can simply be passed through to the graph by deleting all edges derived from that linestring. Then all vertices that were incident to the deleted edges

can also be deleted unless they have incident edges from other linestrings. All points that do have other incident edges have to be analyzed if they became non-essential and have to be removed now.

Conceptually, updates of linestrings can be handles as a sequence of delete and insert operations. No special handling becomes necessary in that case.

Considerations for Graph Algorithms

Graph operations come with a well-established theoretical foundation and a wide variety of algorithms. The mapping of spatial vector data to graphs leads to additional requirements. For example, calculating the shortest path between two points anywhere on a street network usually implies that the two points do not coincide with vertices in the graph. The points could also be located somewhere in between on an edge.

Such issues can usually be addressed by considering both vertices incident to the edge on which the start point lies as start vertices. Likewise, the two vertices of the edge on which the end point is located become the end vertices for the graph algorithm. Four different shortest paths have to be calculated from the start vertex to each end vertex. The results have to be adjusted to include the way from the true start or end point to the respective vertex that was actually used in the graph algorithm. The function *ST_DistanceToPoint* (as presented in [Sto05b]) is applicable there and it should be added to the SQL/MM spatial standard. This function returns the distance of a point on a linestring to the beginning of the linestring. The distance is measured along the linestring. Given that the weight of the edges in the graph is based on the length of the linestrings or rather the respective segments of the linestring, an initial weight for the two start or end vertices can be determined and used in the subsequent graph operation.

The cumulative weights for the four shortest paths are compared to find the truly shortest one. This path is mapped back to the linestrings according to the reverse mapping and the relevant segments of the linestrings are returned as result.

4.3.2 Building Graphs from Point Geometries

Points can be directly mapped to vertices in a graph in the same manner as the points in the definition of linestrings. However, there is no inherent definition for the edges so that the respective application has to determine when points in the spatial table shall be connected in the graph. That highly depends on the application needs. Therefore, an explicit function like *ST_AddEdge* has to be provided. The function takes two points as input parameters and creates an edge in the respective graph. Complementing this function, a routine *ST_AddVertex* should be added to allow the complete construction of a graph from arbitrary, user-supplied point values.

Graphs become named entities or objects in a relational database. Thus, a graph can be solely identified by its name and it does not have to be tied to a (spatial) table and its columns. The two routines can be used to construct and maintain the graph independently if any other data in database.

There are many different ways to derive a graph from points. Two examples are given here to explain the process of the graph construction and the potential benefits of such an implementation. The first scenario connects each vertex in the graph with the vertices corresponding to the nearest points. The second example uses a space partitioning to cluster the points and, thus, the vertices. Both scenarios can be used to efficiently answer nearest neighbor queries.

Connecting n Nearest Neighbors

Nearest neighbor queries can easily be answered with a graph if each vertex in the graph is connected with its n nearest neighbors and the query only requests up to n of the neighbors. The graph additionally stores the distance between the spatial points. The distance is either inherently given with the coordinates being used in the vertices, or it can be stored explicitly as edge weight. The m nearest neighbors (where $m \leq n$) can then be determined with the help of these distances.

Examples for nearest neighbor searches can be found in many fields. Astronomical applications use such facilities to find relations between stars projected on a unit sphere [Sch06]. The stars are represented as single points and the distance between the star and the Earth is not considered. Such queries are particularly interesting if certain regions of the sky and the objects visible in this region are observed and related to each other.

A graph is derived from point geometries in the following way. Each point is mapped directly to a vertex. The key value identifying a point in the spatial table is stored in each vertex as well. Thus, it is possible to relate the results from a graph operation back to the relational data, i. e. the reverse mapping is covered. Then the spatial table is scanned and the n points closest to each point p are selected, together with the respective distance. Directed edges from p to the nearest points are added to the graph. The number n is a fixed number and it establishes the upper limit for edges incident to each vertex in the graph, i. e. each vertex has at most a degree of n.

The nearest neighbors for a point can be quickly determined given that there is a unique mapping from the points to a vertex. If m with $m \leq n$ nearest neighbors are requested for a vertex v (point), the query can be answered by traversing all incident edges of v and sorting them according to their weight. The first m edges lead to the first m adjacent vertices (and points) that represent the desired result.

An issue with this approach is that the upper bound n has to be specified when the graph is constructed. The graph cannot be used when the requested number of nearest

neighbors m exceeds the limit. That is due to the fact that the nearest neighbor relationship is not bi-directional, i. e. if a point p is the nearest neighbor to q, then q may not be the nearest neighbor of p. *Figure 4.16* demonstrates such a scenario with n set to 2. The two nearest neighbors for vertex D are E and F, but E and F do not lead to the third closest neighbor of D, which would be C. Neither has D another outgoing edge so that the complete graph has to be scanned. The situation is even more complex if the fourth closest neighbor of D shall be returned as well. That would be vertex A in the figure and it does not have any edge to D at all.

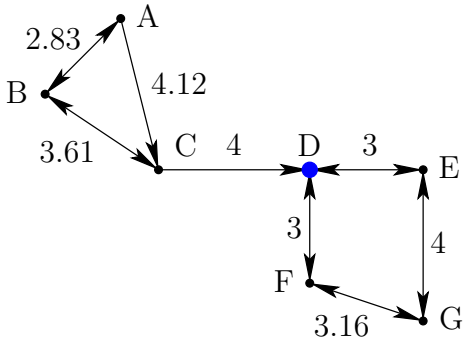


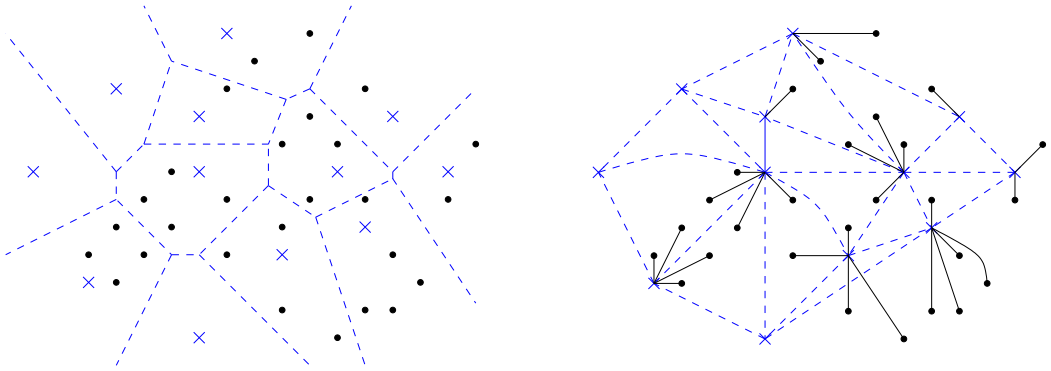
Figure 4.16: Edges to two nearest neighbors

Connecting by Voronoi Regions

A more flexible approach to find nearest neighbors can be achieved by partitioning the space instead of the data. Schmitz [Sch06] uses a Hierarchical Triangular Mesh (HTM) [KST01] for similar, but not graph-based purposes. Voronoi diagrams [BKOS00] are another technique for space tessellations.

The idea is to build a dual graph for the Voronoi diagram, representing each Voronoi cell (Voronoi region) by a vertex in the graph and adding edges between two vertices if their Voronoi cells share a common boundary, the Voronoi edge. So far it is completely independent from the point data in the spatial table, except that the Voronoi cell should be tailored towards the distribution of the data. The actual graph is constructed by finding the Voronoi cell for each point and then connecting the vertex for the point with the representative for the cell. That gives a star-like pattern as in *Figure 4.17*. The representatives and the boundaries of each cell, respective the edges connecting the vertices for the representatives, are drawn as dashed line. The graph is shown as undirected graph but it could be directed as well, depending on the application requirements.

Arbitrary nearest neighbor queries can now be answered with such an approach by starting the query processing in the Voronoi cell of the query argument, i. e. the point for which the nearest neighbors are to be found. The points in that cell are analyzed



(a) Points and Voronoi cell (b) Graph

Figure 4.17: Building graphs based on Voronoi tessellation

and then the neighboring cells are considered. Those neighboring cells are accessible via the edges that connect the cells' representatives. The distances between the query argument and the points in each Voronoi cell are calculated until the required number of neighbors is found and no cell with closer points can be found.

The introduction of Voronoi diagrams for graph construction also allows the exploitation of many other algorithms that were developed for this data structure. Algorithms like finding the closest pair of two points, the computation of a minimum spanning tree, or the calculation of the convex hull can also be supported by graphs based on a Voronoi diagram.

4.3.3 Building Graphs from Polygons

Polygons or collections thereof are the most complex geometries supported by the SQL/MM spatial standard. Similar to points, there is no straight-forward mapping from polygons to graphs. Nevertheless, there are many queries that could take advantage of graph structures built over polygons. However, specialized mappings have to be developed in each case. Some examples are given here to describe the general idea.

Dual Graph for Polygons

Dual graphs are a well-known technique in computational geometry [Kle97]. The idea is to describe a polygon with a single vertex in the graph. The vertices of overlapping polygons or polygons with a common border (i. e. touching polygons) are connected with an edge. A good example are dual graphs for triangulated polygons where a polygon is partitioned with diagonals into several, non-overlapping triangles. The dual graph of the triangulation has a vertex for each triangle and an edge for each diagonal.

Neighbor queries can be answered with dual graphs. For example, if the owner of a real estate wishes to build a new house, local law may require to inform the owners of the neighboring parcels. With the parcels being stored as polygons in a spatial database and a dual graph built for the parcels, the query to find the neighbors boils down to finding the vertex for the parcel on which the activities shall take place and then traverse directly to all adjacent vertices. This traversal will return the key values for the neighboring parcels and any additional information can be retrieved from the relational table.

The direct neighbors could also be determined using the spatial routine *ST_Touches*. However, as soon as several steps are to be followed in the neighbor search, i. e. finding the neighbors of the neighbors, the corresponding query becomes more complicated than the graph traversal, which employs a breadth-first search up to a limited depth.

Another application – also found in the GIS world – could be the supply of several new parcels with electricity or water lines while choosing the shortest route to connect all parcels. A minimum spanning tree algorithm will produce the desired result once each parcel is represented by a vertex in the graph. In case that the exact locations for the end points of the supply lines on the parcels are already known, these end points can be used in the graph to represent the polygons.

With the actual exploitation of the graph functionality being left to the application, a remaining question is how to map polygons to graph vertices. A single point from the interior (or boundary) of the polygon has to be selected. This point is then represented as vertex in the graph in the manner already established in Section 4.3.1 for linestrings. One approach to find that point is availably with the method *ST_PointOnSurface* defined for the ST_Polygon and ST_MultiPolygon types. The SQL/MM spatial standard guarantees that the method is deterministic and, thus, it will always return the same point for a polygon with the same definition. However, a polygon covering a certain area in the real world can be defined in various ways. For example, it is sufficient to define a simple rectangle using five points in the well-known text (WKT) representation (with the first and last point being identical). The same rectangle can also be defined with six or more points if three successive points are collinear. This case is not covered by the standard and, therefore, a homomorphic mapping between the geometry and the graph cannot be relied upon. Using another method like *ST_Centroid* is also not viable since a polygon might have a hole where its centroid is located. Therefore, the point mapped to a vertex in the graph may not lie in the interior of the polygon.

Another issue arises from the fact that polygons can overlap and, regardless of the actual mapping from polygons to vertices, two polygons may result in the same point and vertex in the graph unless preventive steps are taken. *Figure 4.18* illustrates such an example where the point with the smallest X coordinate (and then the smallest Y coordinate) is chosen as reference for the mapping. The red point in Figure 4.18 denotes the point that will be translated to the representative vertex for both polygons. Thus, no unique representative point can be chosen for arbitrary polygons.

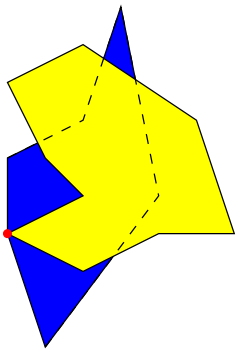


Figure 4.18: Different polygons mapping to same vertex

The consequence is that the key value for each polygon has to be stored in the vertex that represents it. Thus, it is possible to tell the vertices derived from different polygons apart. Even equal geometries are now represented in the graph with distinct vertices. The vertices for equal or overlapping polygons are always connected with an edge so that they become adjacent and are implicitly considered as neighbors in queries. The weight of such edges is set to 0 (zero) to indicate the overlap.

Simplify Polygons to Linestrings

Another approach to deal with polygons with respect to graph functionality is to treat polygons as linestrings and to adopt the facilities described in Section 4.3.1. One way of doing that is to ignore the fact that polygons are surfaces and only take the inner and outer rings. Rings are already linestrings and the methods *ST_ExteriorRing* and *ST_InteriorRingN* can be used to extract the rings from a given polygon.

The second technique to derive linestrings from polygons can be derived from the precision with which real-world objects are modeled in spatial databases. For example, a street may not be stored as a linestring; with a sufficiently high resolution it becomes a polygon truly covering an area. The polygons can be simplified in such scenarios – for example with product-specific extensions to the SQL/MM spatial standard like the *ST_Generalize* routine of the DB2 Spatial Extender [IBM04d].

4.3.4 Impact on Graph Algorithms

Graph algorithms are not impacted by the specific mapping of the different spatial types to graph data structures. The spatial properties inherent to the geometries can be taken advantage of, however, to improve the performance of graph algorithms. Typically, routing calculations or nearest neighbor operations operate on the complete graph. Those operations are expensive for huge graphs; and with spatial data the graph size can grow

quickly, even with the removal of non-essential vertices. For example, the street network of the state Montana consists of 495,042 roads that result in a graph with 372,751 vertices [Bur04]. Performing routing operations between any two points on the roads in Montana requires that all these vertices are involved in the computation with the traditional shortest path algorithms like Dijkstra or Bellman-Ford [CLR01].

The idea is to reduce the work that has to be done by only operating on the vertices that are potentially relevant to the final result. The technique to select a suitable subgraph is already extensively analyzed in the literature [PZMT03, AJ94]. For the particular application of building graphs derived from geometries, the relevant properties are encoded in the vertices, namely the X and Y coordinates. This information can be exploited to terminate the search if the graph algorithm leaves a predefined spatial area. *Figure 4.19* illustrates an example for finding the shortest path from vertex A to B. Any path in the graph leaving the dotted box is not considered any further, assuming that the true shortest path (shown as dashed line) is somewhere inside the box. For example, any path via vertex C is ignored.

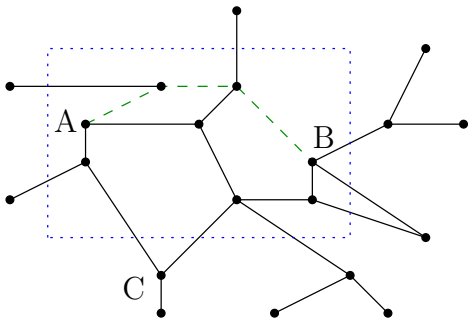


Figure 4.19: Restricting region for shortest path calculation

The test whether a single vertex is relevant can be done by four comparisons that test if the point for this vertex is outside the box. Naturally, the box (usually) cannot be based solely on the start and end points because the shortest path might leave such an area. There would be no shortest path – even no path at all – in Figure 4.19 if only the vertices inside the rectangle spanned by the vertices A and B are used. A good restricting area is as small as possible to reduce the number of vertices considered but it still has to be large enough that the real shortest path is not excluded. The box also depends on the spatial data itself. If there are many edges connecting only a comparably small number of vertices, the box can be made smaller as the chance for the path laying inside a box is higher. With less edges, the box has to cover more space.

For routing operations, it is worthwhile to calculate the (Euclidian) distance between the start and end point for the route. This distance can be used as an additional buffer zone around the rectangle defined by the start and end points. The shortest path could then lead at most the specific distance in the opposite direction, away from the desired end point.

4.4 The Spatial Graph Extender for DB2

The concepts that we described in Section 4.3 are implemented in an extension based on the DB2 Spatial Extender. The so-called Spatial Graph Extender for DB2 UDB is tailored to the specific area to map spatial data to graphs. The extender is built on jGraph [Wit05], a Java-based framework for creating, manipulating, and querying of general graphs in an RDBMS. Zentgraf [Zen06] builds on top of Witzel's work and provides the forward mapping of linestrings. This mapping is even further extended by the reverse mapping and the logic to update the graph if a linestring is added, modified or deleted in the spatial table. The routines *ST_AddVertex* and *ST_AddEdge* together with their counterparts *ST_RemoveVertex* and *ST_RemoveEdge* for the removal of vertices and edges, respectively, are also available to support other geometry types in an application-specific manner.

The goal of [Wit05] was to compare the graph algorithm implemented in SQL using the procedural constructs available with [ISO03k] against a Java [GJSB05] implementation that uses dedicated data structures. As had to be expected, Witzel has shown that the descriptive approach of SQL is not very well suited to answer graph queries that rely heavily on navigational concepts efficiently. The SQL routines are exponentially slower, no matter if procedures or table functions were used [Wit05]. Therefore, only the Java implementation is presented here.

The general requirements for a Spatial Graph Extender are defined in Section 4.4.1 before the architecture and graph functionality is introduced in Section 4.4.2. We present the construction of a graph from linestrings in the Spatial Graph Extender in Section 4.4.3. Two different approaches for the removal of non-essential vertices are discussed. In Section 4.4.4, we explain the automatic synchronization of the graph with data in a spatial table. The details of the spatial routines that take advantage of graphs and operate on them are given in Section 4.4.5. This functionality is also related to the SQL/MM spatial standard as it deviates from the standardized *ST_ShortestPath* function. The meta data managed by the Spatial Graph Extender is very basic as Section 4.4.6 lays out. Finally, we give some performance results for the different approaches to construct a graph in Section 4.4.7. The space consumption of a graph is part of the performance analysis.

4.4.1 Requirements

The Spatial Graph Extender manages the graphs outside (on top of) the database system in its own address space. That approach is necessary because the source code of DB2 is not available. Even if the source code were available, it would require substantial effort to directly integrate the graph logic into the database kernel. Nevertheless, the extender will satisfy the following requirements as well as possible.

Performance & scalability Performance is the single one key requirement for pretty much every database-related functionality. The computation must be fast, even if the scenario is scaled-up to graphs with a large number of vertices and edges.

Concurrency handling for transactions It should be possible that a single graph can be queried by different SQL sessions at the same time. This requirement is mandatory in production environments. Ideally, one expects that the complete graph management happens under transactional control.

Information schema Information describing which graphs exist in the database are to be made available. If graphs are separate objects in a database system, the control of the access to those objects becomes mandatory. The privileges for read access and even more so for modifications on existing graphs are to be controlled.

Backup & recovery Graphs as database objects have to follow the durability principle of the transactional ACID properties. That means, all changes to a graph have to be made permanent beyond the shutdown of the currently running system. The main reason is not to establish full transactional functionality but to build large graphs for reproducible tests.

It is obvious that not all transactional concepts can be preserved and suitable workarounds have to be developed. The main obstacle is that DB2 does not provide any event handlers or triggers that are activated by commit or rollback operations. The transactional semantics cannot be communicated from the database kernel to the extender and the extender cannot guarantee correct transactional behavior.

4.4.2 Architecture

The jGraph Extender consists of several components that are responsible for the management and structure of graphs, maintenance of catalog information, graph operations, access control, and persistency management. These components are executed in a Java Virtual Machine started and controlled by DB2 UDB. That leads to the architecture illustrated in *Figure 4.20*.

The access to the extender is only possible via user-defined routines in the DB2 UDB database system. The routines can be used in SQL statements. The *Graph* component represents graphs using adjacency lists [Wit05, CLR01]. Vertices and edges can be added or removed. All edges incident to a vertex are removed implicitly if a vertex is removed. Vertices are comparable objects, i. e. instance of the Java class *java.lang.Comparable*. Therefore, a hash structure [ML75, Lar78, FNPS79] can be used to provide the direct access to a vertex and its associated adjacency list. The elements in the adjacency list store the weight of the respective edge and the key value of the geometry from which they are derived.

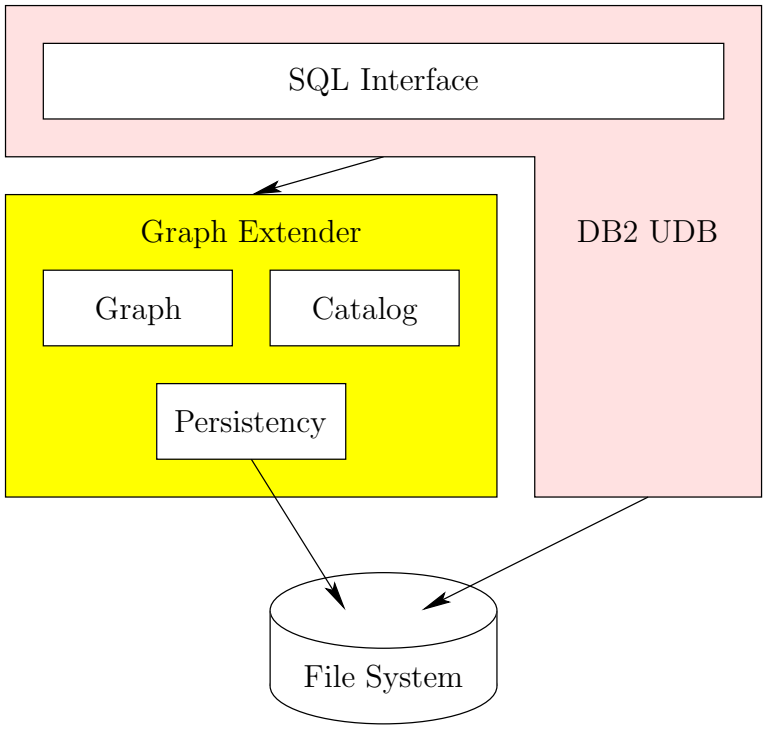


Figure 4.20: Architecture of the Spatial Graph Extender

Graphs may be directed or undirected. The two half-edges for the forward and reverse connection are always maintained – regardless of the directionality – along with a validity flag. That structure allows fast access to the start vertex incident to an edge. Vertex deletions can be sped up because no sequential scan is needed to find the start vertex of an edge in directed graphs if the point to be deleted is the end vertex of the edge.

The graph component also provides methods to query the specific graph. Iterators implement the breadth-first search, depth-first search, minimum spanning tree, and shortest path algorithms. When any of these algorithms is invoked in the graph component, the result is first completely computed and a list generated for it. The algorithm actually returns an iterator to traverse this list.

All graphs are treated like any other database object, for example like tables or constraints. Therefore, the DB2 catalog is extended to describe which graphs were created in the database along with the table that is associated with the graph. The *Catalog* component is dedicated to provide those information. The Spatial Graph Extender includes facilities to control the access to a graph. Therefore, the graph catalog also stores the creator of the graph and which user was granted privileges to query or modify the graph. Additionally, some statistical information like the number of edges and vertices is kept. For consistency reasons, the catalog information is maintained (persistently) in Java only and not copied into a base table in the database system.

The third major component is responsible for the *Persistency* of all graphs. Graphs are in-memory structures only. Each graph is always written completely as a single chunk to the file system. Likewise, whenever a graph is accessed and it is not yet cached, the graph is read from the file system. Its main memory representation is created and then it is cached to avoid excessive file operations.

Process Model

DB2 employs a single Java Virtual Machine (JVM) to execute Java routines. Each routine is run in its own thread and DB2 ensures that the different threads don't interfere with each other by using a specialized class loader as depicted in *Figure 4.21*. The separation is enforced by starting the JVM with a very restricted setting for the CLASSPATH environment variable. The actual classpath set in the database configuration is only considered by the DB2 class loader if the system and bootstrap class loaders cannot find the respective byte code in the directories of the restricted classpath.

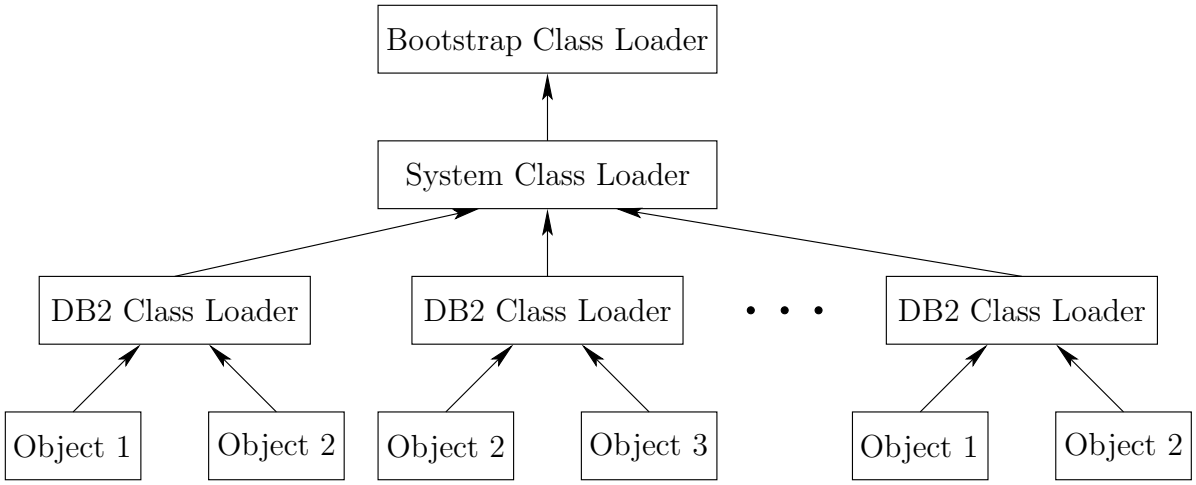


Figure 4.21: Process model of the Spatial Graph Extender

This implementation constitutes a problem for the Spatial Graph Extender because it is not simply possible to access Java objects created by routines executed in one SQL session from routines executed in any other SQL session. Thus, a single instance of a Graph object cannot be simultaneously accessed from concurrent SQL sessions. This issue can be addressed by installing the Java classes of the Spatial Graph Extender as system files or including it with other DB2 Java classes in the restricted classpath that is searched by the system class loader. Once that is done, the *Singleton* design pattern [GHJV95] can be applied and the graph instances and the catalog information can be shared amongst SQL sessions.

4.4.3 Graphs Construction

A graph on a spatial column containing linestrings is constructed with the help of a stored procedure. The procedure *ST_Create_Graph* receives the name of the table and the spatial column as input parameters. Additionally, another parameter might be given to identify the column that provides the information whether the linestring in the respective row is uni-directional or bi-directional.

An empty graph is created and subsequently populated based on the linestrings. The Spatial Graph Extender mandates that the spatial table must have a single key column with a numeric integer data type to uniquely identify each row [Zen06]. The numeric values act as key values that are associated with edges. Once the key column is identified, an SQL statement like the one in *Listing 4.1* is executed to retrieve all the rows and to actually construct the graph. The statement in the listing assumes that the table operated on is named T. The three columns ID, LINE, and DIRECTION identify the column with the key values, the linestrings, and the directionality information, respectively.

```
SELECT id, direction, line..ST_AsBinary(), line..ST_MinX()
FROM   t
ORDER BY line..ST_MinX()
```

Listing 4.1: SQL statement to retrieve linestrings for the graph

Linestrings are converted to their well-known binary (WKB) representation so that the Spatial Graph Extender receives a format it can interpret. The rows in the result set are ordered according to the minimum X coordinate of the linestrings. This ordering is necessary for the in-flight optimization using the *ReduceDuringBuild* algorithm that is described below in more detail. The minimum X coordinate is also retrieved in the select-list for this optimization. Each row is processed in exactly the same fashion:

1. parse the WKB encoding from column LINE,
2. insert each point p_i from the linestring as vertex in the graph (attempts to insert a duplicate vertex are ignored),
3. add an edge for each pair of vertices corresponding to the points p_{i-1} and p_i in both directions, and
4. if the directionality indicates that the linestring represents a uni-directional feature, mark all reverse half-edges just inserted as *non-operative*.

A vertex in the graph is a structure that contains the X and Y coordinate of the point it represents. The comparison operators rely on these coordinates and these operators are needed when vertices are added to or deleted from the graph. Search operations also need comparable objects. Another important task for the X and Y coordinates in the vertex

structure is the reverse mapping for results of graph operations. The point information (i. e. coordinates) is then used to determine the relevant parts of a linestring, for example to identify the start and end point of a segment of a linestring that participates in the shortest path.

An edge is implicitly represented in the adjacency lists. Each list is attached to a specific vertex v and it comprises all vertices adjacent to v. The key value for each edge is kept together with the respective weight as well as the flag whether the edge is operative or not. All three parts of the information are stored twice, once in the forward edge and once in the edge with the opposite direction.

Non-essential vertices (cf. Section 4.3.1) are considered by the Spatial Graph Extender, too. Two different implementations are available to reduce the graph by removing the non-essential vertices. The naive approach, the algorithm *ReduceAfterBuild*, first builds the complete graph as outlined. Then the final step finds all non-essential vertices and removes them. The second algorithm named *ReduceDuringBuild* reduces and optimizes the graph right after each row is processed.

Algorithm ReduceAfterBuild

Building the complete graph up-front including all its non-essential vertices and then reducing it afterwards has the advantage that the construction process is very much streamlined and no overhead is carried along for the in-flight optimization. Thus, the query in Listing 4.1 can be very much simplified by omitting the order-by-clause. Neither is the minimum X coordinate of each linestring needed in the select-list. The execution of the SQL query in the database system will be faster. The complete logic to construct a graph that way consists of the following steps:

1. execute the query
2. for each row:
   a) retrieve the key value and WKB representation of the linestring, and
   b) process the values by adding the vertices and edges to the graph,
3. scan the graph to find all non-essential vertices,
4. bridge each non-essential vertex by directly connecting its two adjacent vertices and then remove the non-essential vertex.

Steps 3 and 4 are separated because the used hash table and list data structures do not provide stable iterators if the underlying structure (the graph) changes. Thus, all non-essential vertices are first detected and collected in a separate list. Once that is completed, the list is traversed and all vertices in the list are removed from the graph.

The graph grows quickly for real-world data with a high number of non-essential vertices and graph operations will take longer because more data has to be searched. The longer search times may out-weight the optimization overhead as Section 4.4.7 demonstrates.

Another issue of this approach is the memory consumption. Each existing non-essential vertex occupies space in main memory because the complete graph structure is kept in main memory. A non-essential vertex has its own adjacency list, usually consisting of two elements only, and the vertex occurs as an element in the adjacency list of at least two other vertices. That are just about 150 bytes per vertex in total, but it adds up with several thousand such vertices. The memory consumption is only an issue until the graph is completely reduced. The final graph always requires the same amount of memory, regardless of the algorithm employed for the optimization.

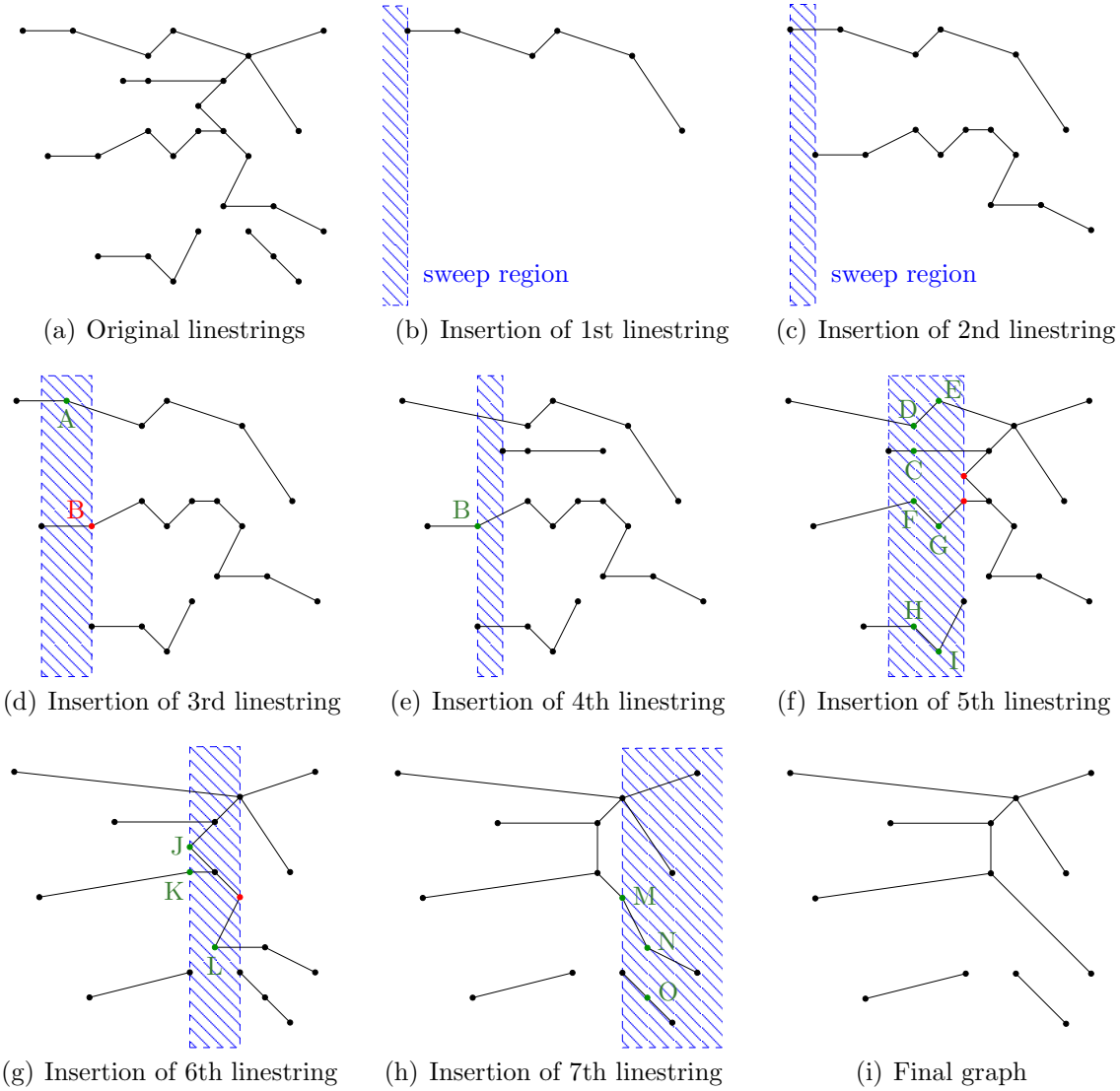
Algorithm ReduceDuringBuild

The performance and the memory consumption during the graph construction can be improved if the reduction is done on the fly, i. e. while vertices and edges are added to the graph. The algorithm *ReduceDuringBuild* implements the improvements by using a sweep line [CLR01] to construct the graph and to reduce it simultaneously.

The sweep line is actually a sweep region and it travels from the smallest X coordinate to the largest one. The region contains those vertices that will be considered for reduction in the current iteration. All vertices derived from points with larger X coordinates than the sweep region are already or may still become vertices of intersections when new linestrings are mapped to the graph. All additional linestrings are always added to the right of the sweep region. All vertices to the left of the sweep region, i. e. the vertices from points with smaller X coordinates, are already completely processed and no new intersections may occur in that area. When a linestring is to be processed and added to the graph, the sweep region is bound by the minimum X coordinate value of the previous linestring and the minimum X coordinate of the current linestring. The minimum X coordinates of the linestrings are monotonously growing because the query to retrieve the linestrings from the spatial database orders the result set ascending according to the minimum X coordinates as Listing 4.1 showed.

The algorithm *ReduceDuringBuild* is illustrated in *Figure 4.22*. Figure 4.22(a) shows the linestrings for which a graph is to be built. Initially, the graph is empty and does not contain any vertices or edges. Subsequently, the linestrings are handled in the order of their minimum X coordinate. The sweep region for the insertion of the third linestring (Figure 4.22(d)) contains the first non-essential vertex (named A), which is removed right away. The vertex B is exactly on the right boundary of the sweep region. It cannot be removed because there is no guarantee that another linestring with the same minimum X coordinate will be processed next and produce an intersection right at this point and its vertex. The vertex is removed in the next step, however, when it is guaranteed

that no such intersection exists (Figure 4.22(e)). With the insertion of the last, seventh linestring, the graph is nearly completed. The remaining non-essential vertices M, N, and O are removed and the final graph is built as in Figure 4.22(i).



(a) Original linestrings (b) Insertion of 1st linestring (c) Insertion of 2nd linestring

(d) Insertion of 3rd linestring (e) Insertion of 4th linestring (f) Insertion of 5th linestring

(g) Insertion of 6th linestring (h) Insertion of 7th linestring (i) Final graph

Figure 4.22: Constructing graph with algorithm *ReduceDuringBuild*

The data structure to find all vertices in the sweep region during the optimization phase following the insertion of a linestring is essential for good performance of the sweep algorithm. A priority queue is used for that purpose. The entries in the priority queue are the vertices, ordered by their X coordinate. When a linestring is processed, all its essential and non-essential vertices are inserted into the priority queue. The priority queue is consulted to retrieve all points in the sweep region until a point is encountered that has an X coordinate that is larger or equal to the right boundary of the sweep region.

The procedure to construct the graph G and reduce it on the fly implements *ReduceDuringBuild* as shown in *Algorithm 2*. The details regarding the initialization and finalization of the process as well as the sweep region and the priority queue *pq* are also defined. The algorithm receives the identifying name of the relational table it operates on, along with the name of the spatial column that contains the linestrings. The left boundary of the sweep region is not explicitly stored in a variable. Instead, the minimum X coordinate of the first element in the priority queue represents that boundary implicitly. Note that the *getMinVertex* operation on the priority queue retrieves the vertex with the minimum X coordinate but it does not remove it from the queue.

Algorithm 2 *ReduceDuringBuild(spatialTable, spatialColumn, graphName)*

```
1: G ← empty graph
2: pq ← ∅
3: rightSweepBound = −∞
   /* build graph */
4: resultSet ← query spatial table as in Listing 4.1
5: while fetch row from resultSet into id, direction, wkb, xMin do
6:    vertices ← map points in wkb
7:    edges ← map line segments between points in wkb and associated id
8:    G.insert(vertices)
9:    G.insert(edges) /* forward and backward edges, honoring direction flag */
10:   pq.insert(vertices)
      /* reduce graph in sweep region */
11:   rightSweepBound ← xMin
12:   while pq.size() > 0 do
13:      v ← pg.getMinVertex()
14:      if v.getXCoordinate() ≥ rightSweepBound then
15:         break
16:      else
17:         pq.deleteMin()
18:         G.optimizeVertex(v)
19:      end if
20:   end while
21: end while
   /* reduce graph in final sweep region */
22: while pq.size() > 0 do
23:   v ← pg.getMinVertex()
24:   pq.deleteMin()
25:   G.optimizeVertex(v)
26: end while
```

The routine *optimizeVertex* determines if the given vertex is non-essential according to Definition 2. If it is, new edges connecting its two adjacent vertices are inserted in the graph and the vertex v is removed together with its incident edges.

4.4.4 Updating Graphs

The graph reduction poses a problem when data modifications are applied to the linestrings from which a graph is constructed. In Section 4.3.1, we already described the general approach for graph updates by reconstructing the graph in the affected area, including all non-essential vertices. The new or changed linestring is then mapped to the graph before the area is reduced again. A routine *UpdateGraph* is implemented in the Spatial Graph Extender to add a new linestring to an existing graph. *Algorithm 3* shows the logic of the routine.

Algorithm 3 *UpdateGraph(graphName, id, direction, line)*

1: $G \leftarrow$ graph identified by $graphName$
2: $(spatialTable, spatialColumn) \leftarrow$ table and column on which G was created
3: $pq \leftarrow \emptyset$
   / insert new linestring */*
4: *vertices* ← map points in *line*
5: *edges* ← map line segments between points in *line* and associated *id*
6: G.insert(*vertices*)
7: G.insert(*edges*) */* forward and backward edges, honoring direction flag */*
   / reconstruct non-essential vertices in affected area */*
8: *resultSet* ← query *spatialTable* to retrieve all linestrings in *spatialColumn* that intersect *line*
9: **while** fetch row from *resultSet* into *id*, *direction*, *wkb*, *xMin* **do**
10: G.deleteEdge(*id*)
11: *vertices* ← map points in *wkb*
12: *edges* ← map line segments between points in *wkb* and associated *id*
13: G.insert(*vertices*)
14: G.insert(*edges*) */* forward and backward edges, honoring direction flag */*
15: *pq*.insert(*vertices*)
16: **end while**
   / reduce graph in affected area */*
17: **while** *pq*.size() > 0 **do**
18: $v \leftarrow pg$.getMinVertex()
19: *pq*.deleteMin()
20: G.optimizeVertex(v)
21: **end while**

The routine is available in the spatial database as a user-defined function. The function is invoked automatically by a trigger for every data modification statement that adds a new linestring. Deletions of a linestring are simple to handle using the graph routine *deleteEdge* and by subsequently removing all vertices that became non-essential during the edge removal. `UPDATE` statements are treated as a pair of delete and insert operations. The triggers for all three operations are created in the procedure that also builds the graph itself. Thus, the user does not have to take care of the graph maintenance manually and the graph is synchronized with the geometry data in the spatial table.

4.4.5 Querying Graphs in SQL Statements

The construction and maintenance of a graph as described in Sections 4.4.3 and 4.4.4 are mandatory features of the Spatial Graph Extender. However, the real benefit of graphs only comes from the facilities to query and evaluate the graph. The Spatial Graph Extender implements a table function *ST_ShortestPath* that takes the start and end point along with the identifying name of the graph as input. (In that it differs from the standardized function that receives a multi-set of linestring values.) It returns a table with three columns where one column is the sequence number in the shortest path, another column contains the (part of the) linestring that participates in the shortest path and the last column is used for the key values identifying the row with the respective linestring.

An SQL statement that calculates the shortest path between two points in a graph named *SAMPLE_GRAPH* is shown in *Listing 4.2*. The graph is created for table T with which the result is joined in order to extract more detailed information on the path segments, for example the name of the street. The final result is ordered by the sequence of the segments in the shortest path.

```
SELECT sp.line, t.street_name
FROM   TABLE ( ST_ShortestPath(ST_Point(3.25, 3.25),
                    ST_Point(5.5, 6.5), 'SAMPLE_GRAPH') )
          AS sp(seq, line, id) JOIN t ON sp.id = t.id
ORDER BY sp.seq
```

Listing 4.2: Query to find shortest path between two points

The linestrings from which the sample graph is derived are depicted in *Figure 4.23*. The start point s and the end point e between which the shortest path (set as dashed line) is determined are also shown.

The processing in the *ST_ShortestPath* routine takes the following steps. The first two steps provide the forward mapping from the given start and end points to the respective edges in the graph. The Steps 3 and 4 are required to handle the cases where the start and end point do not coincide with vertices in the graph and, thus, up to four possibilities have to be evaluated in Step 5. The final two steps implement the reverse mapping to return the pieces of the original linestring geometries that belong to the shortest path.

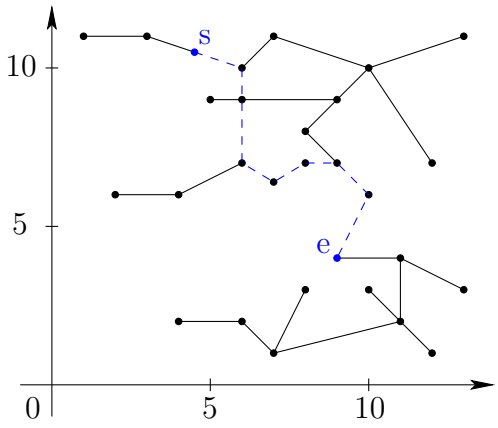


Figure 4.23: Sample linestrings for shortest path computation

1. Find the linestrings l_s and l_e on which the start point s and the end point e lie using the *ST_Intersects* method.

2. Determine the distances of s and e from the beginning of l_s and l_e, respectively, using the routine *ST_DistanceToPoint* that was described in [Sto05b].

3. In the graph, find the vertex for the first point of l_s. Follow the edges of l_s and sum up the weights until the edge is found where the total weight exceeds the distance calculated for s for the first time. Let v_{s1} and v_{s2} be the two vertices incident to this edge. Do the same for l_e to determine v_{e1} and v_{e2}.

4. Let w_{s1} and w_{s2} be the initial weights associated with v_{s1} and v_{s2}, respectively. w_{s1} is the distance from the beginning of l_s to s minus the distance from the beginning of l_s to the point represented by v_{s1}, and w_{s2} is the distance to the point represented by v_{s2} reduced by the distance to s. The initial weights w_{e1} and w_{e2} for the vertices v_{e1} and v_{e2} are calculated likewise.

5. Determine the four shortest paths between the two start vertices v_{s1} and v_{s2} and the two end vertices v_{e1} and v_{e2}, adding the initial weights for the respective start and end vertex. The final shortest path is the one with the lowest weight of all four paths. Any shortest path algorithm can be chosen for that as long as it returns a sequence of edges that form the desired path.

6. For each edge r in the shortest path, determine the incident vertices and extract the coordinates for both to construct the corresponding points p_{r1} and p_{r2}.

7. Trim the linestring (identified by the key value stored in r) by removing all points prior to p_{r1} and after p_{r2} using a new method *ST_ExtractSegment* that is defined for linestrings and multi-linestrings. The remaining piece of the linestring is returned together with the key value *id* and a sequence number.

The SQL/MM spatial standard does not provide any functions like *ST_ExtractSegment* to perform the necessary trimming in Step 7. The method *ST_MeasureBetween* of the DB2 Spatial Extender [IBM04d] goes into this direction as does the Linear Referencing Feature in Oracle Spatial [Ora05f]. However, both products use M coordinates to identify portions of a linestring. Another approach would be to use the *ST_DeleteVertex* of the Informix Spatial DataBlade [IFX02b] or the similar method *ST_RemovePoint* provided by the DB2 Spatial Extender. However, those routines only remove a single point, still requiring additional logic. Therefore, the Spatial Graph Extender retrieves the well-known binary (WKB) representation of the linestrings and modifies the WKB directly. The WKB is traversed until the first point p_{r1} is found. This point and all subsequent ones are copied to a new WKB until the second point p_{r2} is encountered. The new WKB is given to the database system to construct a linestring geometry.

The bottom line of the presented approach is that the true shortest path including all non-essential points appears in the result of the SQL query from Listing 4.2. Besides the specification of the graph name, the caller does not have to be aware at all that the actual operation is not directly performed on the linestrings. A deeper integration of graph functionality into the database kernel would even do away with the parameter that identifies the graph to the *ST_ShortestPath* function. Instead, the name of the spatial column that stores the linestrings could be used and the system can identify the graph based on that through its database catalog. Thus, graphs would become real index mechanisms and could be used transparently. We discuss this issue in more detail in Section 4.5.

4.4.6 Graph Information Schema

A graph is constructed for the linestrings stored in a base table in a DB2 database. The table functions used to query a graph expect the identifying name of the graph as input. Thus, a relationship exists between a graph name and a spatial column. This information is maintained in the Graph Information Schema (also known as graph catalog) along with some additional information. The graph information schema is very basic and it only consists of two views whose relational schema is shown in *Listing 4.3*. The underlined columns mark those columns that uniquely identify a record in the respective view.

```
graphs ( graph_name, creator, create_time, change_time,
         table_schema, table_name, spatial_column_name,
         number_of_vertices, number_of_edges )

graphauth ( graph_name, grantee, grantor, controlauth,
            selectauth, insertauth, deleteauth )
```

Listing 4.3: Views of the graph information schema

The first view is named GRAPHS. It shows information about the available graphs, their creator, creation time as well as statistical information on the number of edges and vertices in the graph. The three columns TABLE_SCHEMA, TABLE_NAME, and SPATIAL_COLUMN_NAME uniquely identify the column from which the graph is derived. If no table is associated with the graph and it was built with the explicit routines *ST_AddVertex* and *ST_AddEdge* only, the three values will be NULL. Otherwise, the view ST_GEOMETRY_COLUMNS of the SQL/MM spatial information schema can be consulted for information on the spatial column.

The Spatial Graph Extender supports privileges on graphs and the view GRAPHAUTH provides the information which user or group may query or modify a graph. The names of the privileges are closely aligned to the privileges available in SQL. Privileges can be held or not and they can also be held *with grant option*, which has the usual meaning. The routines *ST_Graph_Grant* and *ST_Graph_Revoke* are implemented in the Spatial Graph Extender as stored procedures to provide an interface to manage the privileges.

Internally, data presented by both information schema views is exclusively kept in the Java code of the Spatial Graph Extender. Table functions are implemented to retrieve the information from the extender and to represent it as relational table. The advantages are fast access during graph processing and no inconsistencies due to duplicated storage of the information in the database system and in the graphs. The durability of the information is ensured by the *Persistency* component of the extender.

4.4.7 Performance Evaluation

The Spatial Graph Extender was used to measure different performance aspects during the graph construction. Especially the removal of non-essential vertices was evaluated based on real-world data to be able to judge the effectiveness of our approach. Graph operations like the calculation of minimum spanning trees or shortest path queries are not deeply analyzed because of the main memory implementation of the extender. Real-world applications will have to use a more sophisticated storage mechanism as Section 4.5.1 describes.

The street network of the single states of the United States of America provides the test data. The data originates from the Dynamap/1000 database [Tel06] and it was collected in the year 1998. Its shapefiles [ESR98] were obtained from the Bureau of Transportation Statistics that is part of the U.S. Department of Transportation [Bur04]. Each state has different characteristics of its streets and also a different number of streets, ranging from just 14,762 streets in the District of Columbia to 2,259,314 in Texas. Thus, the performance tests can cover a wide range of graph sizes. The streets of California are shown in *Figure 4.24* by means of an example. It shows a high density of streets in most regions of the state.

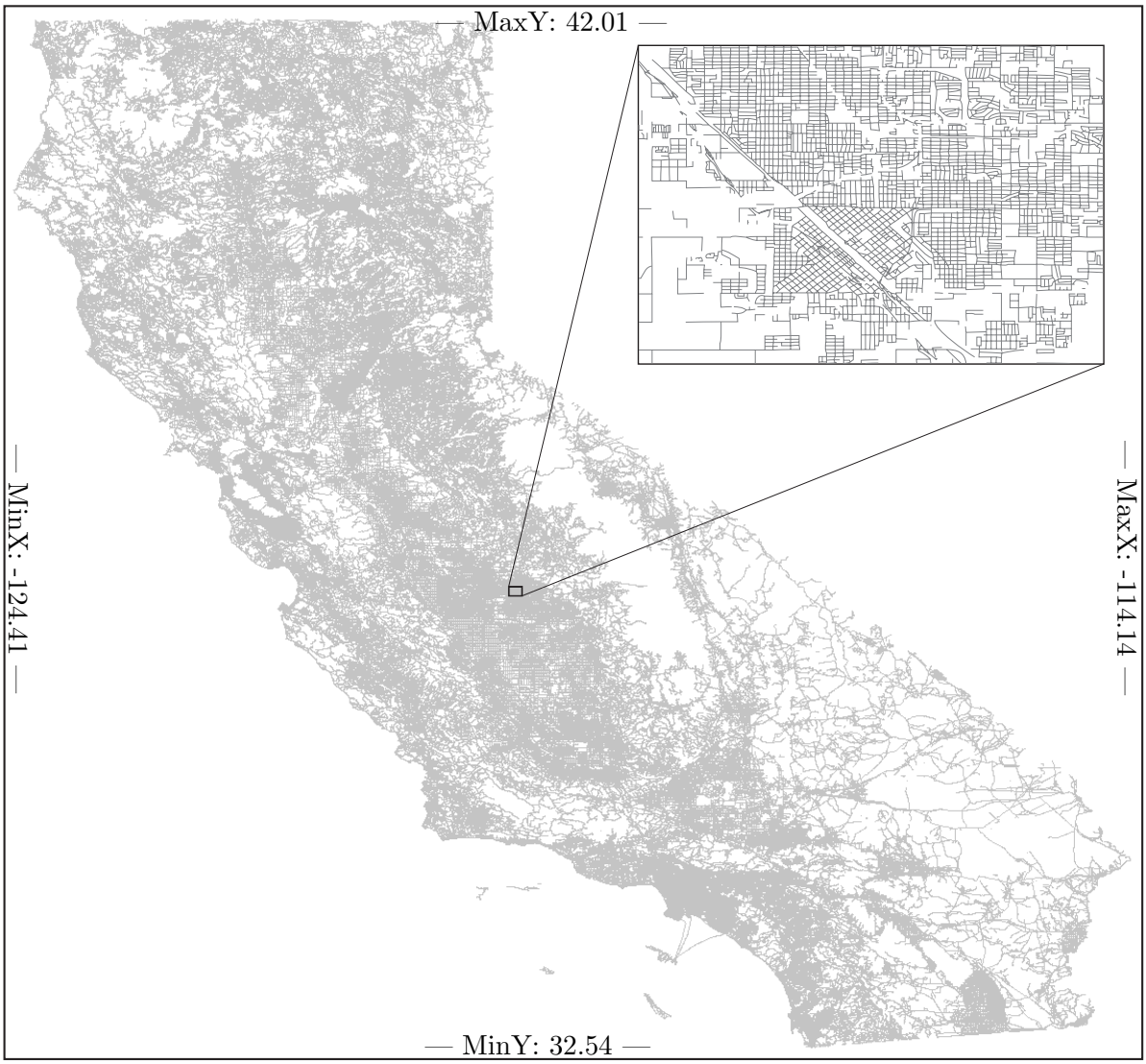


Figure 4.24: Visualization of streets in California

The street network of the eight states California, Connecticut, Delaware, New Mexico, Michigan, Pennsylvania, Texas, and Wyoming are used to build a graph for each and to apply graph operations on them. *Table 4.1* shows exactly how many streets are available for each state, how many essential and non-essential vertices each graph is comprised of and how many vertices are non-essential.

State	number of linestrings	total number of vertices	number of non-essential vertices	portion of non-essential vertices
California	1,589,712	4,867,271	3,666,488	75.3%
Connecticut	148,232	340,099	227,489	66.8%
Delaware	39,429	91,673	62,407	68.1%
District of Columbia	14,762	14,059	4,975	35.4%
Michigan	595,662	1,361,034	932,599	68.5%
New Mexico	451,326	2,248,623	1,917,837	85.3%
Pennsylvania	842,099	2,469,506	1,849,249	74.9%
Texas	2,259,314	7,214,599	5,525,320	76.6%
Wyoming	270,334	1,926,298	1,724,727	89.5%

Table 4.1: Test data for graphs

An IBM pSeries 7025-6F1 serves as test system. It comes with 8 GB main memory and sufficient disk space on SCSI hard drives to hold the operating system (AIX Version 5.2), DB2 UDB Version 8.2.4, the spatial data in shapefiles and the graphs constructed from the spatial data. Four Power-IV CPUs with 600 MHz each provide the necessary processing power.

Memory Consumption

A crucial piece for graph operations is the amount of main memory occupied by a single graph. Main memory is a very scarce resource. It has to be used carefully since all processes running on the machine have to share it. This constraint also applies to graphs and especially to the construction of a graph based on a set of linestrings because a smaller graph can be operated on more quickly, resulting in faster response times. One reason for the considerations of non-essential vertices is to reduce the amount of memory required to store a graph.

The Spatial Graph Extender is implemented using the Java programming language. Given that and the actual data structures used to represent graphs it can be derived that a single element in the adjacency list for a vertex occupies 56 bytes. The average space requirement for a single vertex in a graph is about 160 bytes as *Table 4.2* shows.

These 160 bytes include the overhead to maintain the adjacency lists and the hash table where each adjacency list is hung off. The total size of the graph was measured by serializing and writing it to a file as a byte stream using the persistency component. The resulting file size is then divided by the number of vertices in the graph. The sizes for the full and the reduced graphs are shown to illustrate the effect of the removal of the non-essential vertices.

State	size of full graph (in MB)	size of reduced graph (in MB)	average size per vertex (in bytes)
California	750.1	218.6	162
Connecticut	53.4	20.4	165
Delaware	14.5	5.4	165
District of Columbia	2.7	2.0	200
Michigan	216.4	81.2	167
New Mexico	339.8	61.7	158
Pennsylvania	383.4	115.3	163
Texas	1111.1	310.1	161
Wyoming	287.1	37.1	156

Table 4.2: Space requirements of graphs

The growth of the graph size is approximately linear, depending on the number of line-strings processed already. *Figure 4.25* illustrates the behavior for the construction of the graph for the state of Michigan. The two algorithms *ReduceAfterBuild* and *ReduceDuringBuild* are included. The *ReduceAfterBuild* algorithm does not show an exact linear behavior. That is due to the actual data being processed. The first $\frac{3}{4}$ of the tuples contain only short streets with few intersections and short adjacency lists. The remaining streets are longer leading to more intersections and longer adjacency lists. The graph size grows more quickly for those streets. The order of the rows retrieved from the database system is the same as the order of the data in the shapefile that was imported into the database. The query used in the Spatial Graph Extender does not imply any reordering and DB2 resorts to a table scan so that the rows are returned in the same order as they were inserted. The algorithm *ReduceAfterBuild* has a different behavior for the memory consumption if the tuples are sorted by their minimum X coordinate prior to being inserted into the graph. A strictly linear growth is observed now.

At the very end of the construction, i. e. when all rows from the table are processed, the reduced graph always has the same size, regardless of the chosen algorithm. However, the algorithm *ReduceAfterBuild* has a much larger memory footprint during the construction itself. That is due to the fact that all the non-essential vertices are kept until the

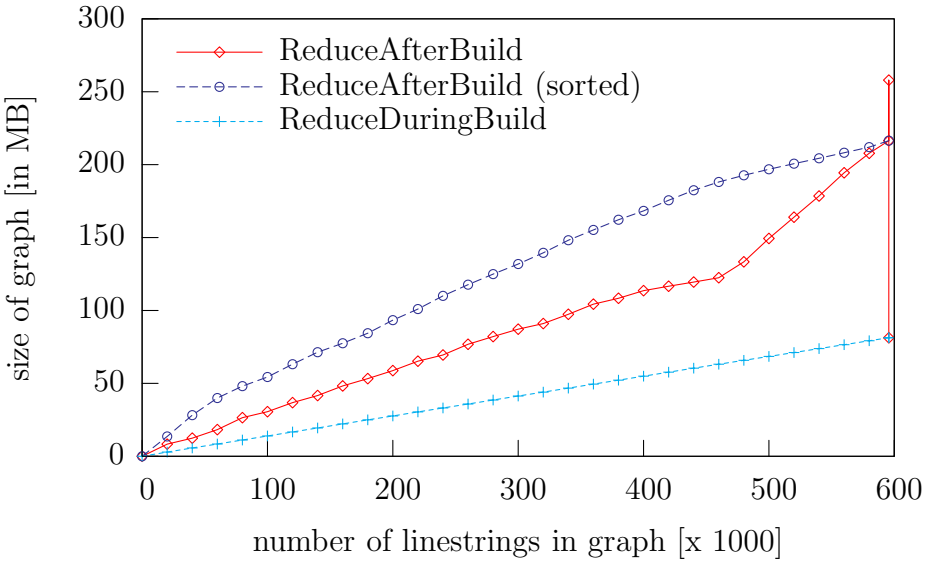


Figure 4.25: Memory consumption during graph construction for Michigan streets

reduction process starts. Additionally, a peak occurs when all linestrings are added to the graph and a temporary data structure is needed to collect all non-essential vertices that are subsequently removed. The *ReduceDuringBuild* removes those vertices in between and reuses their space when needed. Another data structure (the priority queue) is now required, but the queue adds marginal overhead of about 100 KB only.

Other states show similar behavior for the memory consumption. The gap between the two algorithms may be narrower or wider, depending on the number of non-essential vertices in the respective graphs. There is a direct, proportional relationship between the number of essential vertices, the final graph size and the size of the graph during the construction process. It should be noted again that real-world street networks are used and other, artifically constructed data sets may yield other results. Thus, the results may not be applicable to other data sets.

Execution Time for Graph Construction

The memory footprint for graphs is one essential criteria for a deployment of graph functionality in enterprise applications. Another, usually even more important performance aspect is execution time. The execution time of the graph construction process is evaluated for both algorithms implemented.

The execution time to construct the graphs for the various states is compared in *Figure 4.26*. The figure shows the total time to construct and reduce the respective graph. The algorithm *ReduceDuringBuild* has a distinctive advantage for large graphs, even if it requires the additional sorting of the linestrings in the relational table based on

their minimum X coordinate. This ordering adds overhead, but the intermittent reduction performed by the sweep region out-weighs the overhead beginning at approximately 250,000 linestrings. *ReduceAfterBuild* is faster for smaller graphs if the sorting is omitted. For comparison, the construction of the graphs based on the sorted linestrings is also included for the *ReduceAfterBuild*. The figure illustrates that the time difference between both algorithms can be solely attributed to the sort operation for small graphs. The intermittent reductions do not add more overhead than the late graph optimization of *ReduceAfterBuild*.

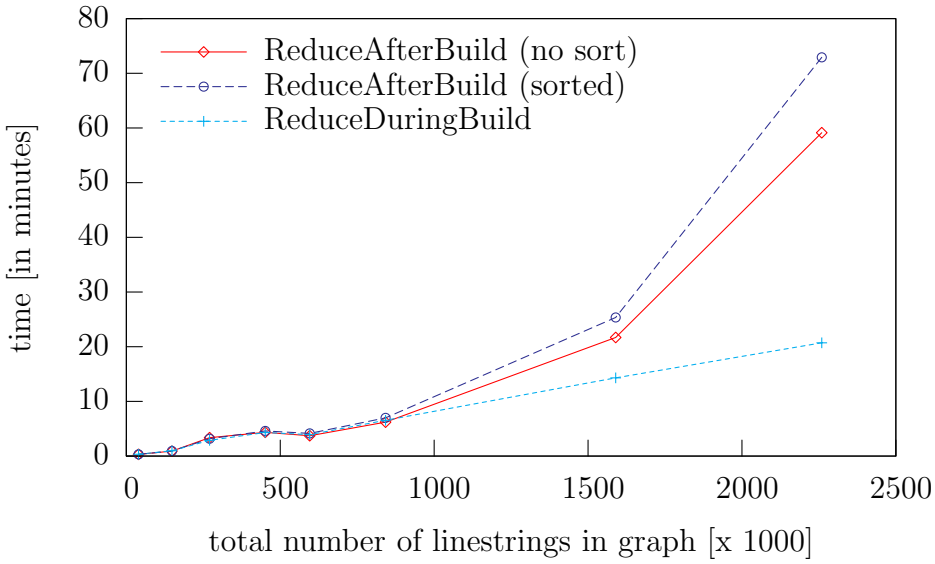


Figure 4.26: Time to construct a graph

The overhead of the sort operation necessary for the *ReduceDuringBuild* algorithm is also clearly visible when the times for the separate phases during the graph construction are summarized. *Figure 4.27* is based on the 451,326 streets of the state New Mexico. It shows the growth of the cumulative time needed for each of the phases. First, an initialization happens when the SQL query is built to retrieve the linestrings and their associated key values from the spatial relational table. The query is compiled by the DB2 server and an instance of the Java class *ResultSet* created. About 65 milliseconds are needed for the compilation. Subsequently, each linestring and key value is *fetched* via the result set. The WKB encoding of the linestring is *parsed*, the single vertices for the graph constructed and *inserted* into the graph along with the edges derived from the linestring. The final step performed at the end of the processing of each linestring is to *optimize* all vertices in the sweep region, i. e. to remove the non-essential ones. As can be seen in the figure, there is a high initial cost of 26 seconds already attached to the fetch operation for the very first linestring. The major part of this time is spent by DB2 to sort the linestrings by their minimum X coordinates.

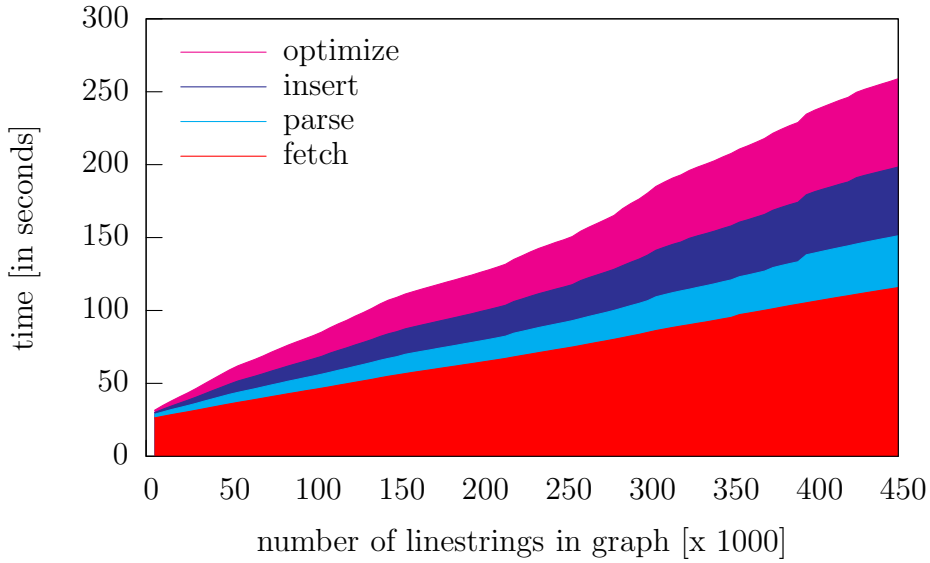


Figure 4.27: Time portions of phases during graph construction for New Mexico

All of the phases scale linearly for the algorithm *ReduceDuringBuild* when the graph grows. The proportions between the single phases for the state of New Mexico are summarized in *Table 4.3*. The total construction time on the test system amounts to 260 seconds. Other data sets can lead to varying behaviors because the linestrings could be shorter and, thus, the time to be spend during parsing and optimization is reduced. Zentgraf [Zen06] shows that with measurements of other sets of linestring data. Another observation is that there is no single function in the Spatial Graph Extender that would benefit from specific optimizations. The fetch phase – as the most time consuming step – is completely dominated by the processing inside the DB2 engine and the DB2 JDBC driver; the logic in the Spatial Graph Extender is minimal. All other phases are roughly equal.

Phase	Cumulative Time	Portion
Initialization	65 ms	0%
Fetch	116.0 s	44.7%
Parse	35.6 s	13.7%
Insert	47.3 s	18.2%
Optimize	60.7 s	23.4%

Table 4.3: Portions of phases for graph construction for New Mexico streets

For comparison, building a B-Tree index on a VARCHAR(30) column in the same table with the New Mexico streets takes already 20 seconds. Naturally, the DB2 code is heavily optimized for a fast index creation. It also uses C++ code that has a distinct

advantage over Java. Finally, the index manager has internal access to the data pages of the table and does not have to pass through other layers in the DB2 engine like the language binding API. Therefore, it is quite acceptable that the graph construction is just one magnitude slower.

4.4.8 Conclusions for the Spatial Graph Extender

The Spatial Graph Extender shows that the integration of graph technology with spatial vector data is feasible. It allows the direct support for routing operations and other graph algorithms on geometry data by providing the result of the operation through a table function that can be joined with any other data in the database.

We do consider the performance of the Spatial Graph Extender as being quite acceptable. A graph derived from 1.6 million linestrings (like the ones for the state of California) is built in under 15 minutes using the *ReduceDuringBuild* algorithm. This approach also comes with the smallest maximum memory footprint (of the implemented algorithms) during the construction process with 218.8 MB.

A native implementation of graph functionality internally in the database kernel is very much preferable over the Spatial Graph Extender, however. Requirements like tight transactional integration is not available in the extender. In particular, trigger-based maintenance of the graph during insert, update, or delete operations on the relational table from which the graph was derived will not notice it if a transaction is rolled back. The graph is maintained in a memory area external to the database system and no mechanisms are available to notify the extender of the rollback operation, not to mention that the Spatial Graph Extender does not even keep track of transactions and does not ensure the isolation properties. Thus, it cannot undo any changes.

Integrated backup and recovery of graphs is also not supported by the extender. All graphs are materialized by writing the Java objects to a file on disk. A dedicated routine of the extender is invoked when the Java Virtual Machine is being shut down or an explicit operation to make the graph persistent is called. That leaves the task to collect the graphs with the backup image to the database administrator.

A native implementation written in a programming language like C or C++ also promises a further performance improvement due to the language and its features. Additionally, the memory footprint can be reduced even more because the overhead carried along by Java to identify its objects can be done away with. A single element in the adjacency lists occupies 56 bytes in Java and a C implementation can store the same information in 44 bytes (16 bytes for the coordinates identifying the vertex, 4 bytes for the integer key value of the linestring, 8 bytes for the weight, and another 16 bytes for 64-bit pointers to the next and previous elements in the adjacency list). A reduced memory consumption also implies faster operations with growing graphs.

The current situation is that graph functionality has not been implemented as a built-in feature of the various commercial and open source database systems. Therefore, the Spatial Graph Extender is currently still the best available option.

4.5 Integrating Graphs into a DBMS

The jGraph Extender in [Wit05] and its enhancements to the Spatial Graph Extender successfully demonstrate that the integration of graph functionality in relational database systems is feasible and a very useful feature to have, particularly in the context of spatial database systems. However, an important argument against the extender is the approach to hold graphs completely in main memory. Although this technique is accepted and products adopted similar ways, for example the DB2 Net.Search Extender [IBM04g] keeps a full text index in main memory, a deeper integration into the database system is usually desirable and sometimes even mandatory.

We discuss in Section 4.5.1 how main memory graph structures can be broken down onto database pages. These pages can then be managed like any other database page, e. g. data pages or B-Tree index pages, in the buffer pool of the DBMS. The second desirable feature for a full integration of graph functionality into the database kernel is to treat graphs as another kind of indexes. We go into those detail in Section 4.5.2.

4.5.1 Graph Storage

Graphs that are derived from spatial data inherit the spatial properties of the geometries by way of the forward and reverse mapping. These properties can be exploited to break down the graph into smaller, connected pieces. Once the pieces are small enough, a graph can simply be stored on database pages. This topic is already discussed extensively in research literature. We summarize some of those findings.

Several external-memory graph algorithms are described in [CGG+95, FMS98]. The foundation is laid for algorithms operating on graphs too big to be contained in main memory. The overhead for the disk I/O operations in a block-based manner is taken into consideration. Cost models for graph algorithms are developed based. Goodrich [NGV93] also fits into this category of prior work, but he is only concerned with the block-based storage to accommodate graph traversal problems. Goodrich does not address other computations and algorithms on such stored graphs.

The Connectivity-Clustered Access Method (CCAM) [SL97] detects clusters in a graph based on the connectivity between vertices. Single clusters are stored together in so-called *connectivity-clustered data files* that use a B^+-Tree for the access and fixed-size data pages for the information associated with each vertex. Thus, a mapping to pages used in relational database systems is implicitly given. CCAM does not only consider a

one-time clustering of the graph but is also concerned with the graph maintenance and dynamic reclustering. Algorithms operating on graphs are adjusted to take into account the clustered storage. With an adjustment to the clustering criteria, these results can be carried over to graphs derived from spatial vector data. The adjustment exploits the spatial relationship between vertices.

A hierarchical partitioning of graphs is proposed by [JHR96]. Jing describes an approach to find the shortest paths in a network by using the Hierarchical Encoded Path View (HEPV). The imposed hierarchy is a hybrid between main memory (memory-resident) graphs and disk-based algorithms. Jing integrates both worlds in an efficient manner. The shortest paths on higher levels in the hierarchy are precomputed and, thus, the necessary work during query time can be constrained to subgraphs on the lower levels. Additionally, the hierarchy comes with an implicit partitioning of the graph that allows it to be stored on the relatively small data pages used by a DBMS.

Hub-indexing [GSVGM98] is a technique similar to the hierarchical partitioning. Goldman uses a specialized index structure to quickly compute the distance between two vertices in a graph. The index contains so-called *hub vertices* that connect clusters in the graph. The shortest path between vertices in different clusters always has to cross such a hub vertex. The shortest path from a vertex in a cluster is computed to all the hub vertices of the cluster to avoid the exact computation of the shortest path at query time by properly combining the precomputed information. Another side effect is again that the clustering implies a break-down of the graph into smaller pieces that are containable in the data pages in a DBMS.

Liu [LSC94] presents an overview of various clustering or partitioning strategies for graphs, including CCAM and other spatial access methods. In particular, the use of traditional spatial indexing techniques like the cell tree [GB91] or space filling curves [Sag94] are compared with each other with respect to their performance. The employment of existing and well-established spatial indexing techniques as an access method to the vertices of a graph is very appealing, especially if the vertices of the graph are derived from geometries already. For example, an R-Tree [Gut84] implements a data partitioning that can be applied to the vertices and their associated adjacency lists.

Today's relational database systems provide other mechanisms that could be exploited for the storage of graphs. The multi-dimensional clustering (MDC) as implemented by DB2 UDB [IBM04e] is intended for data warehouse applications and it establishes a convenient means to store graphs. Using ranges for the X and Y dimensions to define a space partitioning causes each vertex in the graph to be assigned to a cell of the partitioning, based on the coordinates of the points from which the vertex is derived. The vertex and its adjacency list can now be stored on the data pages assigned to the cell in MDC index. The MDC feature is already tailored to accommodate dynamic growth once the allocated space for a cell fills up. No further considerations for the cells with a high occupation will be necessary from that perspective.

4.5.2 Graphs as Index Structures

The function of the Spatial Graph Extender that make the graph operations (including the necessary reverse mapping) available to applications require that the name of the graph is given as input parameter in order to identify the graph on which to operate. That is necessary because the external routines written in Java are not aware of the origin of the spatial data they operate on, i. e. the routines do not have the information of the table and its spatial column. This issue is addressed by the extender with the identification of the graph by its name.

Very much preferable is to treat graphs transparently like index structures. To that end, the external routines would be invoked in SQL statements as shown in *Listing 4.4*. The spatial column is used for the parameter of a graph routine and the database system automatically rewrites the query in such a way that the corresponding name of the graph is determined internally with the help of the database catalog and provided to the external code as was done manually in Listing 4.2. Naturally, it does require awareness of the SQL compiler and optimizer of the DBMS.

```
SELECT ...
FROM   t, TABLE ( ST_ShortestPath(ST_Point(3.25, 3.25),
              ST_Point(5.5, 6.5), t.spatial_column) ) AS sp
WHERE  t.id = sp.id AND ...
```

Listing 4.4: SQL statement to use graphs like indexes

As a side effect, this syntax allows the construction of the graph on the fly if no graph index exists – based on the geometries given as input. In that case, the semantics of the *ST_ShortestPath* function are the ones of an aggregate function because all of the geometries have to be processed to construct the graph before any result can be returned, i. e. before the shortest path can be computed.

Such an implementation would be similar to the scalar function *Contains* that is part of the SQL/MM full text standard [ISO03c]. However, for graphs it is not a scalar function because the shortest path, for instance, is a table comprised of the sequence number identifying the traversal order, the actual part of the linestring, and the key value of the linestring that is traversed.

4.6 Summary

Graph functionality is essential in spatial database systems as it provides the basis for routing-based operations and nearest neighbor queries, for instance. Such operations are not only important in GIS scenarios, but a wide array of other applications can benefit as well, in particular any situation where spatial networks are relevant. We explained in this chapter how graph functionality can be added seamlessly to the SQL/MM spatial standard [ISO03d].

The basic concepts of graphs were introduced. That provided the base line for the subsequent mapping of geometries to graphs. The mapping of linestrings or multi-linestrings is straight-forward and each point in the linestring definitions becomes a vertex in the graph. The edges are derived from the line segments connecting those points. Our mapping consisted not only of a forward mapping from the set of linestrings to the graph, but it also covers the reverse mapping that is necessary to identify the linestrings (or parts thereof) in the spatial table that correspond to the result produced by a graph algorithm. Thus, algorithms like the shortest path resort to the graph transparently and the user invoking the respective function deals with geometries only.

A specialty of linestring geometries is that not all points in the definition of a linestring are relevant in the graph. Some of the points are non-essential, i. e. they connect only two line segments from the same linestring. Removing the corresponding vertices does not impact the graph algorithm from a functional point of view, but it results in noticeable performance improvements due to smaller graphs.

The mapping of point geometries and polygons to graphs require additional, domain-specific information regarding the connectivity in the graph. Points become vertices but the remaining question is how to connect vertices. Two different approaches were presented. We determine the n nearest neighbors for each point and an edge to each of them is added to the graph. Another mechanism employs a space partitioning and connects the points in one cell of the partition with a representative of the cell. Neighboring cells are connected as well. Alternatively, the user can create the edges between the vertices manually. We handle polygons in a similar fashion, unless a geometry can be reduced to a linestring if possible.

We implemented these concepts for the forward and reverse mapping of linestrings in the Spatial Graph Extender. The extender is written in Java and it keeps the complete graph in main memory. We have shown that the removal of the non-essential vertices from the graph results in substantial benefits in real-world street networks in terms of smaller memory footprints as well as faster construction of the graph itself.

The gaps left by the Spatial Graph Extender, most notably the adjustment of the storage model for the graphs, have to be closed in order to pave the way for a future integration of graph functionality into the kernel of relational database systems. To that end, it is necessary to split the graph into smaller pieces so that it can be stored on the data pages used by the DBMS. Additionally, the graph algorithms in the extender are currently in-memory algorithms and adjustments are necessary to take the external storage of parts of the graph into account. Some work towards this goal can already be found in the literature, but the spatial properties of the vertices may offer additional possibilities.

5 Support for Three-Dimensional Data

An important aspect of spatial data is the dimensionality of the geometries and the dimensionality of the data space. The majority of the open source and commercial products to manage spatial data as first class citizens in relational database systems available today only support two-dimensional data. Even the SQL/MM spatial standard is targeted at two dimensions only. For various applications, most prominently computer-aided design (CAD) systems, these two dimensions are by far not sufficient; at least a third dimension is needed. For example, designing a car or an airplane requires three-dimensional data [Pöt01].

This chapter is dedicated to lift this restriction in a seamless way. We describe how the standardized spatial type hierarchy can be extended to support three-dimensional objects. Section 5.1 gives an overview on the already standardized aspects for handling three-dimensional data and their current shortcomings. We also present a few examples where three dimensions are needed in applications. Section 5.2 proposes the extension of the standardized spatial type hierarchy in detail and which methods should be provided for 3D-related types. The results of the extension on the usability are analyzed. Section 5.3 is focuses on with the extension of the external data formats well-known text (WKT) and well-known binary (WKB) (cf. Section 2.3.3). We specify and evaluate two different approaches for the external formats. The meta data for spatial data is maintained in the spatial information schema defined by the SQL/MM spatial standard. Section 5.4 discusses which influence the addition of three-dimensional data will have on that information schema. We implement a prototype based on the DB2 Spatial Extender to prove that the previously described support for three-dimensional data is actually feasible. The findings of that work, including results of performance measurements, are given in Section 5.5 before we close the chapter with a summary in Section 5.6.

5.1 Motivation

Today's spatial extensions for relational database systems are tailored to support only two-dimensional data and operations. Some products deviate from that by allowing the storage of coordinates for a third and even fourth dimension in points of geometries [IFX02b, Ora05f]. The third dimension are the Z coordinates, and the fourth dimension are for the so-called measures or M coordinates. However, those Z and M coordinates

are not exploited for spatial operations like comparing or overlapping geometries. Sometimes, the approach to carry the coordinates for the additional dimensions is called to be a 2.5-dimensional model. Effectively, the products only provide a two-dimensional model.

In many situations it is necessary to not only consider two dimensions for spatial operations, but also take a third dimension into account, e. g. the height. For example, the field of architecture or city planning requires true three-dimensional data and operations. Data about bridges or tunnels need to be stored very detailed for an application concerned with transportation issues. Another area where higher dimensional spatial data is mandatory are CAD applications [HL93]. Computer-aided design is not related to geographic information systems (GISs), which are still the traditional area for spatial extensions in database systems. CAD applications do indeed benefit from spatial database systems. However, modern aircrafts or cars are built from literally millions of different parts. Each part must be designed. The spatial properties (especially the shape) of each part must be available for verification purposes, for example to test that all parts fit properly together and that no two parts fill the same space in the final product [Pöt01].

Given that the roots of the SQL/MM spatial standard [ISO03d] lie in the GIS area, it is not surprising that the current standard only defines an interface for two dimensions. The current working draft (as of May 2006) for the third version of the SQL/MM spatial standard [ISO05a] began with the addition of 3D support. Facilities to store Z and M coordinates can be maintained in geometries – as it is already implemented in various products. The working draft is not yet an international standard, and there appears to be no schedule when the draft shall be progressed to finally become an international standard. Furthermore, standards development is a dynamic process and it may very well come to pass that the addition of Z and M coordinates will be removed before too long. Nevertheless, adding support for a third dimension is the next logical step for further developing the standard. The remainder of this chapter proposes one approach for such an extension.

The enhancement of the standardized spatial data handling shall adhere to the following requirements:

- seamlessly support spatial data in $\mathbb{R}^2$ and $\mathbb{R}^3$, including dedicated data types for true three-dimensional geometries,
- all spatial operations currently defined in the SQL/MM spatial standard shall be supported with an adequate meaning in $\mathbb{R}^3$, and
- include support for different two- and three-dimensional spatial reference systems (SRSs).

The proposed extension is based on realistic requirements and it does not attempt to enable the full scope of every imaginable three-dimensional object and operations on such objects. Instead, the current spatial products are analyzed. For example, only the restricted support for polygons with linear boundaries is carried over to $\mathbb{R}^3$. If additional requirements for curved surfaces arise and if productized spatial extensions pick up those requirements, the SQL/MM spatial standard should be further enhanced by defining interfaces for such objects as well.

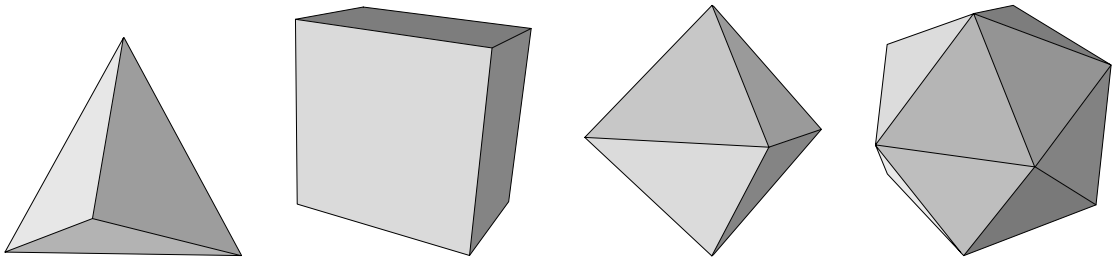
5.2 Spatial Type Hierarchy and Methods

The addition of real three-dimensional data requires an adequate modeling of the new geometry types in the spatial type hierarchy. A representation of three-dimensional objects is introduced in Section 5.2.1. In Section 5.2.2 we describe the modifications to the hierarchy that defined in [ISO03d]. Previously, Section 2.3.2 discussed the problems with the standardized type hierarchy and presented an alternative solution. We add data types for 3D geometries in Section 5.2.2 to that modified hierarchy. Once the data types are defined, Section 5.2.4 explains the actual functional enhancements. We describe the problems with the current 2D semantics of the spatial methods and the necessary changes to address those. We also introduce new and additional methods specific to 3D data.

5.2.1 Representing Objects in Three-Dimensional Space

Brisson [Bri90] describes various ways to represent geometric objects in higher-dimensional space. A straight-forward and intuitive extension of the standardized spatial type hierarchy as defined in the SQL/MM spatial standard is the addition of polyhedra. A polyhedron as a three-dimensional object can be defined by a series of polygons in 3D space. Using this approach follows closely the paradigm to define an n-dimensional object by a set of objects of dimension $n - 1$. Specifically, a polygon is comprised of at least one linear ring, which in itself is defined by a set of points. Thus, moving this scheme to the next dimension is obvious.

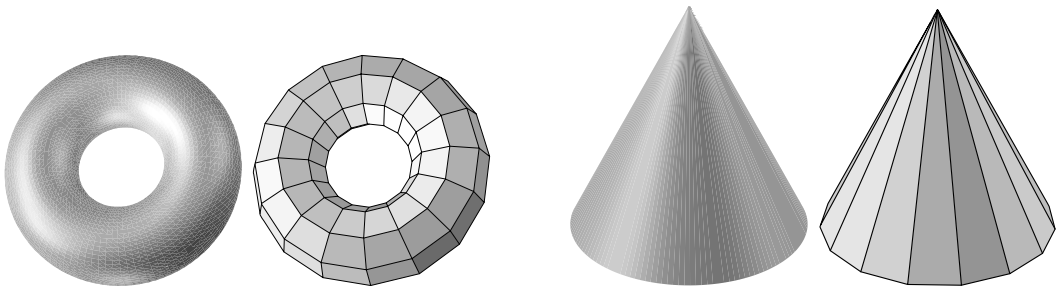
Polyhedra can be described by multiple polygons, also called facets. Each facet represents one planar piece of the surface of a three-dimensional object. *Figure 5.1* shows a few typical examples of polyhedra. The tetrahedron in Figure 5.1(a) is the simplest three-dimensional object defined by only four points. It consists of six edges and four surface areas. The other Figures 5.1(b) thru 5.1(d) depict other, more complex polyhedra. Of course, these are just examples and by far not an exhaustive list of polyhedra. More complex three-dimensional objects can be constructed by combining tetrahedra, for example.



(a) Tetrahedron (b) Cuboid (c) Octahedron (d) Icosahedron

Figure 5.1: Examples for polyhedra

Unfortunately, it is not possible to represent *all* three-dimensional objects exactly by using polyhedra. *Figure 5.2* shows two examples. All objects with a surface that is not comprised of planar polygons only cannot be described. These kinds of objects have generally curved surfaces. The SQL/MM spatial standard accommodates for such geometries in $\mathbb{R}^2$ and defines various curved types, e. g. ST_CurvePolygon. If curved surfaces for 3D objects are not supported by a product, the accepted technique is to approximate the geometry by linear or planar geometries that do not have arbitrary curves. The Figure 5.2 also shows the respective approximation.



(a) Torus (b) Cone

Figure 5.2: Examples for non-polyhedra and their approximation

Due to the already very high complexity for managing with polyhedra, including 3D operations, the following sections always assume that any non-polyhedron geometries is approximated by a polyhedron. Like the SQL/MM spatial data types to manage circular curves or curved polygons, additional types for three-dimensional objects with curved surfaces can be added to the type hierarchy if the requirement comes up. Such types should be made optional for conformance.

There are two alternatives to describe a polyhedron. The first follows the approach known from the triangulation of polygons [BKOS00] and applies that technique to $\mathbb{R}^3$ by partitioning a polyhedron into tetrahedra, the so-called *tetrahedronization* technique.

Such an approach is implemented in the TEN system (TEtrahedral Network) [ZRS02] as data model for the spatial information. *Figure 5.3* illustrates the steps how a simple cube can be partitioned.

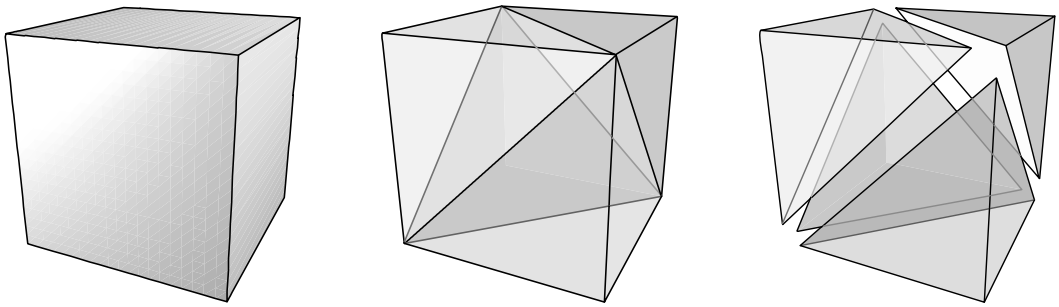


Figure 5.3: Partitioning a cube into tetrahedra

Although a simple inductive proof shows that each polygon can be triangulated using only the vertices of the polygon as the vertices of the triangles, this result cannot be applied directly to the three-dimensional space. There are some polyhedra that cannot be partitioned by solely using the vertices of the polyhedron as the vertices of the tetrahedra and additional points in the interior are needed for the successful partitioning [BKOS00]. That situation complicates the representation of arbitrary polyhedra using the union of tetrahedra; sometimes additional points are needed for constructing a tetrahedron.

The second representation follows more closely the descriptive style adopted by the SQL/MM spatial standard for linestrings, polygons, and collections. Polyhedra are defined by specifying and by grouping the set of polygons to form the surface of the object. A polyhedron has one or more boundary surfaces, named *shells*. The surface defines the boundary, which separates the interior and the exterior of the polyhedron. The single polygons must be planar and all polygons define together the complete surface of the object. A valid polyhedron defined that way has to be closed, i. e. every edge at the object surface has a polygon to its right and as well as to its left side. Additionally, every non-trivial vertex is incident to at least three facets [Haj70]. (A trivial vertex is collinear with two other vertices on the same edge, or it is a vertex on the interior of an edge that connects two facets, which are on the same plane.) *Figure 5.4* shows a valid and an invalid surface of an approximated sphere. All mentioned conditions are met in Figure 5.4(a). A part of the sphere surface was removed in the second case (Figure 5.4(b)), leaving a hole and an invalid surface description for the polyhedron. The edges of the facets at the cut-off only have a facet to the left and not to the right side (assuming a counter-clockwise traversal of the edges belonging to a facet). Additionally, not all vertices are incident to at least three facets, namely the vertices at the cut-off belong to two facets only.

(a) valid (b) invalid

Figure 5.4: Valid and invalid polyhedra

5.2.2 Extending the SQL/MM Type Hierarchy

The SQL type hierarchy as standardized in the SQL/MM Part 3: Spatial standard [ISO03d] is based on the class hierarchy defined by the OpenGIS Simple Features Specification for SQL (SFS) [OGC99]. Compared to the simple feature specification, the major difference in the SQL/MM spatial standard can be found in a set of additional optional types to handle circular curves by themselves or as (part of the) boundaries of polygons. However, those extensions were not consistently adopted because the circular curves and curved polygons exist only as primitive types and not as collection types as Section 2.3.2 already explained. Handling of three-dimensional data was neither added to the Open Geospatial Consortium (OGC) class hierarchy nor to the SQL/MM type hierarchy. Of course, the Open Geospatial Consortium is aware of the requirement to handle a third dimension, and the current Version 3.1 of the Geography Markup Language (GML) includes a XML type SolidType and related types. *Figure 5.5* depicts an excerpt type hierarchy defined with the eXtensible Markup Language (XML) as it is specified in [OGC04] and [ISO05b]. It shows only the types related to the three-dimensional data. The XML schema name gml is omitted in the figure because it applies to all types in GML.

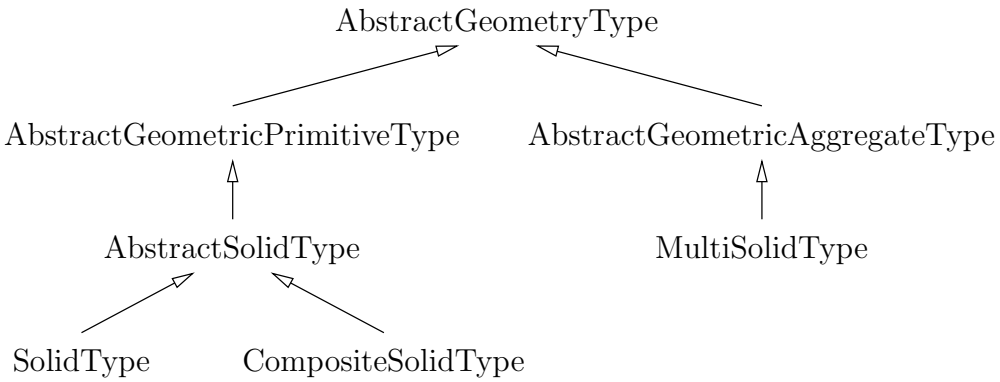


Figure 5.5: 3D types in GML type hierarchy

The basic concepts of the 3D types in the GML specification can be carried to the SQL/MM spatial standard. A close observation of AbstractSolidType shows that this GML type does not add any further information to AbstractGeometricPrimitiveType, and AbstractGeometricPrimitiveType itself is effectively identical to AbstractGeometryType. Therefore, those two types are not to be introduced in the 3D extensions for the SQL/MM spatial standard. Instead, the addition of the types ST_Solid and ST_MultiSolid is a more adequate and fitting extension. This enhancement is shown in *Figure 5.6*. The ST_ prefix is used to follow the naming conventions of the standard. Polyhedra are very common and widely used three-dimensional representations, and we intent to support polyhedra directly. Therefore, the additional data types ST_Polyhedron and ST_MultiPolyhedron are defined as subtypes of solids and multi-solids, respectively. Other subtypes to model cones, cylinders, or spheres with their curved surfaces could be handled by products implementing the standard. The type ST_Solid is not instantiable (abstract) and it is derived from ST_Geometry. It corresponds to the geometric primitive Solid in GML. Its purpose is to represent any arbitrary three-dimensional object. ST_Polyhedron is an instantiable subtype of ST_Solid to represent objects with a surface that consists of polygons only. The SQL standard [ISO03i] does not provide any means to implement the generic composite design pattern [GHJV95] for structured types because SQL does not have the concept of pointers or references like C/C++, Java or XML. Therefore, the additional instantiable type ST_MultiPolyhedron is needed for collections of polyhedra.

Every three-dimensional object can be defined by the shells describing its boundary. The first shell is the exterior boundary of the object, and all other shells define holes in the geometry. For consistency reasons with polygons, the holes have to be in the interior defined by the exterior shell and they shall not overlap. *Figure 5.7* shows a cube that has a single hole in its interior. It has one exterior shell and one interior. For a better visualization, the cube is also shown cut open. According to the definition, a torus like the one in Figure 5.2(a) does not have a hole because a single exterior shell can be used to describe it.

A geometric aggregate in GML is represented as a geometry collection in SQL/MM. Thus, the ST_MultiSolid is introduced to represent collections of three-dimensional objects that shall be treated as a single, scalar value in a relational database. The single solids in such a multi-solid may touch at the boundary, and the touching region must not exceed the dimension of 1 (one), i. e. only points or edges. Furthermore, the interiors of the solids must not overlap.

The SQL/MM spatial standard defines in its conformance clause which features an implementation of the standard has to provide in order to claim conformance. Today, the major target area of implementations of the SQL/MM spatial standard are two-dimensional applications, primarily GISs. Therefore, making the 3D extension a mandatory requirement for standard conformance is not an option as it would break the conformance of implementations that do adhere to the standard now. Instead, a new

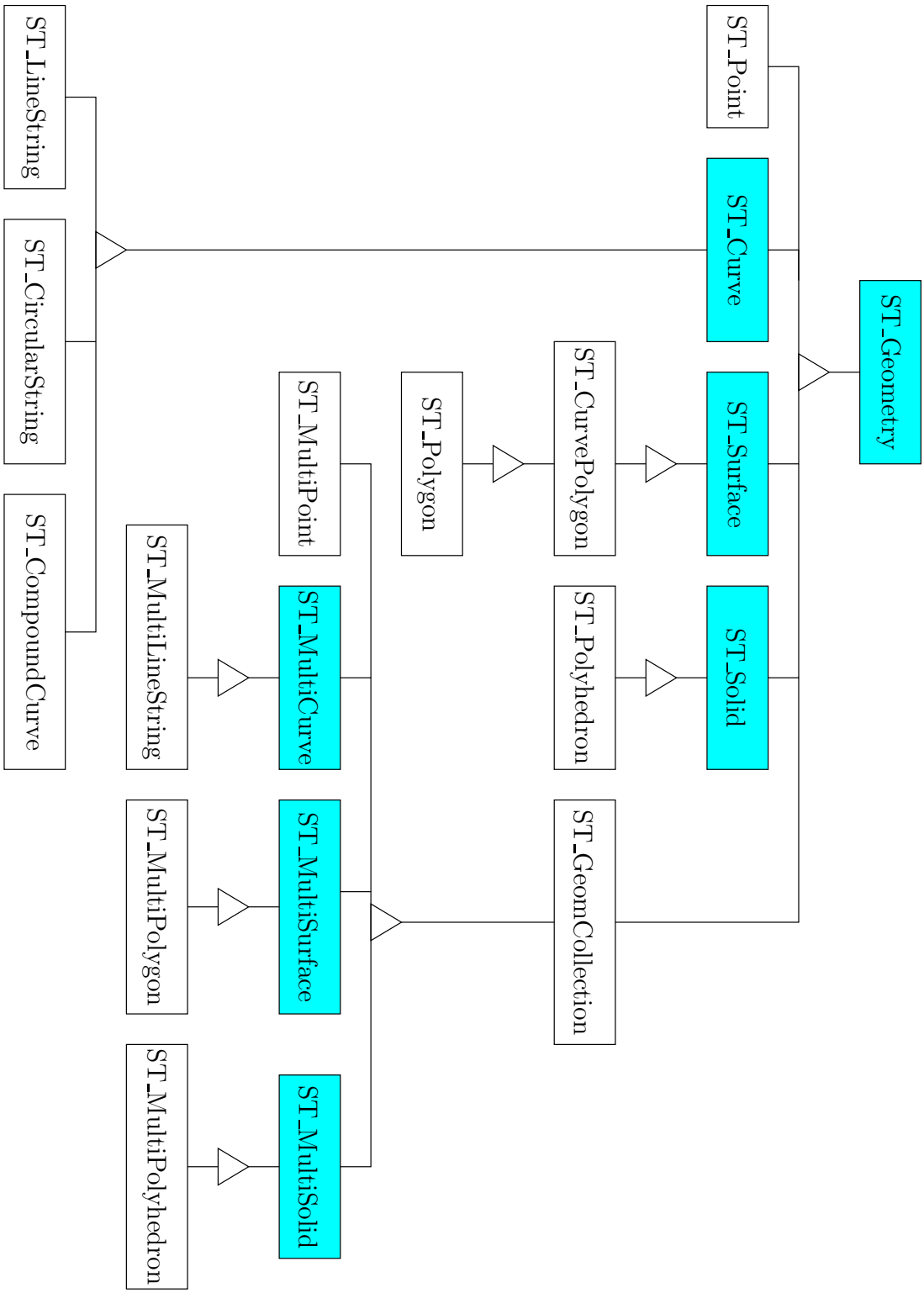


Figure 5.6: SQL/MM spatial type hierarchy with 3D types

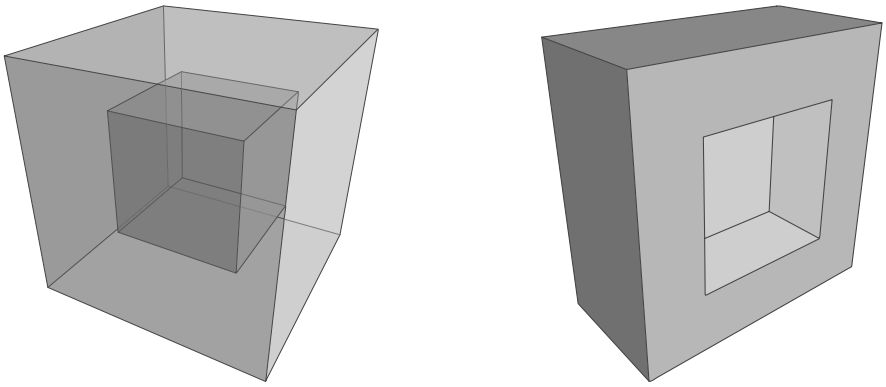


Figure 5.7: Cube with a hole

feature like "3D feature" should collect the four additional types and their related functionality. If an implementation of the standard provides support for three-dimensional objects, it can claim conformance to the SQL/MM spatial standard, including the 3D feature.

The new types include a series of methods applicable to data in 3D. We describe those new methods in detail in Section 5.2.4. The methods apply to the modified hierarchy from Section 2.3.8 as well, even if not all types are present.

5.2.3 Extending the Modified Type Hierarchy

We already mentioned in Section 2.3.2 that the spatial type hierarchy in the OpenGIS Simple Features Specification for SQL (SFS) was obviously designed with the implementation in mind. The SFS does not deeply consider the user who just wants to use the data types. We developed an alternative modeling that focused on the alignment of the types in the hierarchy according to the geometric properties in Section 2.3.8. As a result, the type ST_GeomCollection merged with ST_Geometry and the subtree for the collections disappeared. The previously duplicated functionality now appears only at a single place in the hierarchy. The type ST_Empty exists to represent empty geometries and, thus, the special handling for empty points, empty linestrings, empty polygons and the like is avoided.

The issues raised in Section 2.3.8 apply to the extension for three-dimensional objects presented before. Again, separate processing needs to be performed for collections of 3D objects and just single 3D objects. That overly complicating the use of the spatial data types. Furthermore, semantically similar methods are duplicated in both branches of the hierarchy, i. e. the same method has to be declared for geometric primitives and for collections.

Figure 5.8 shows the modified type hierarchy and how intuitive its extension for the third dimension becomes. The difference to the type hierarchy in Figure 5.6 with respect to the three-dimensional data types is the omission of ST_Solid; the other new types can be found directly, although in a different relationship. The hierarchy should be extended below ST_MultiSolid if additional types for three-dimensional objects shall be supported besides polyhedra, for instance spheres or cylinders.

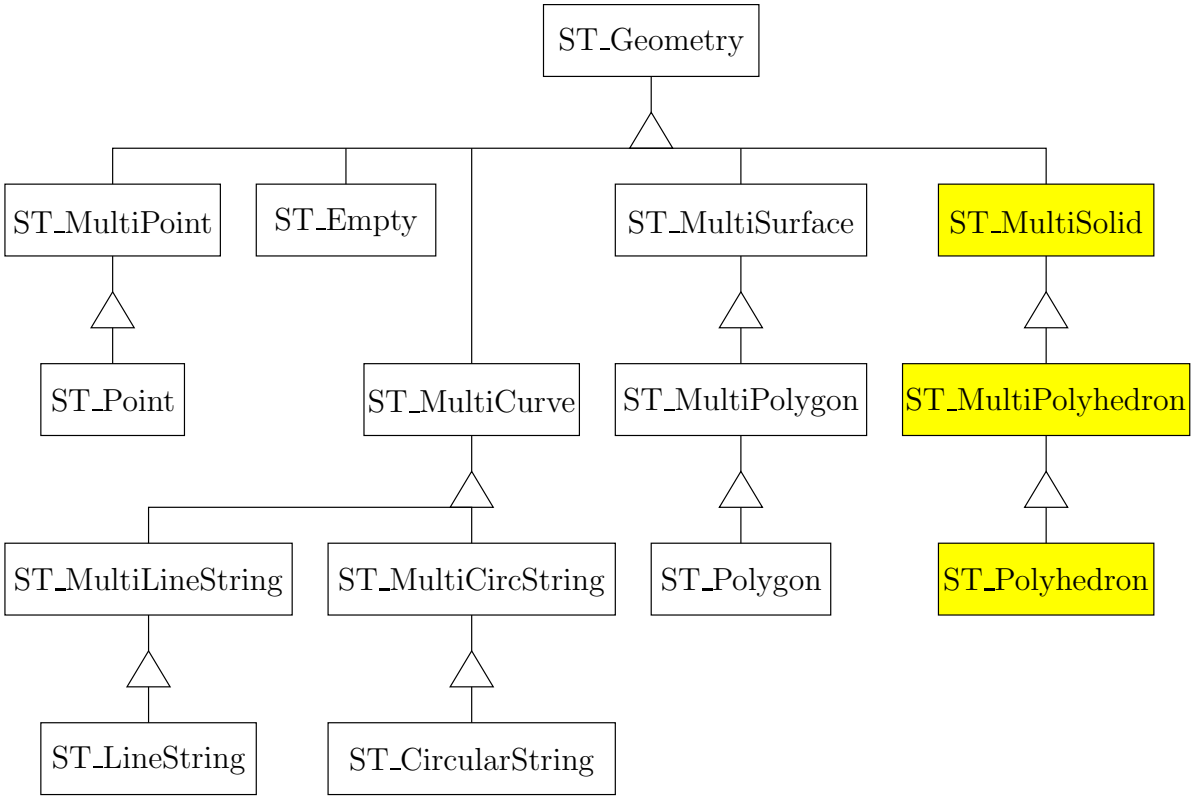


Figure 5.8: Modified spatial type hierarchy with 3D types

5.2.4 Processing 3D Data in Spatial Methods

The actual functionality of spatial extensions needs to be implemented in the methods and functions that operate on the types of the spatial type hierarchy. There are, of course, new methods associated with the types that model three-dimensional spatial objects. These are described first.

The move to handle three-dimensional data in general also has an impact on the existing routines. For example, a point could be placed in the three-dimensional space by defining its X, Y, and Z coordinate. Calculations with that point could now take place in $\mathbb{R}^3$. The

effect of the 3D enhancements to existing methods declared for the 0-, 1- or 2-dimensional types (i. e. points, curves, and surfaces, respectively) is discussed afterwards.

New methods

The move to three-dimensional data is very similar to the move from one-dimensional data (curves) to two-dimensional structures (surfaces) in terms of specific methods that should be provided for the new data types. Those new methods are concerned with the properties that come with the additional dimension, i. e. an object has now a surface as boundary, consists of multiple shells that make up the boundary, has a volume and the like. That leads to the definition of the following methods for the types ST_Solid and ST_MultiSolid. The names suggested for the methods are restricted to a length of 18 characters so that a requirement for the feature F391, "Long identifiers", of [ISO03i] is not imposed.

ST_Volume returns the volume measurement of the solid. If the geometry is comprised of several parts, i. e. it is a multi-solid, then the volume of each (non-overlapping) part is computed and the final result is the sum of the volumes of the single parts. A unit of measure may be specified optionally.

ST_BoundingArea returns the area measurement of the boundary of the solid or multi-solid. In case of a multi-solid, the bounding area is calculated for each solid and the results are summed up. A unit of measure may be specified optionally.

ST_Centroid calculates an arbitrary point in 3D space that is the mathematical centroid of the geometry. Such points can be used as reference points for markings in print-outs, for example.

ST_PointInSolid calculates a point that is anywhere in the interior of the solid or multi-solid. The intersection of that point with the boundary or exterior of the geometry is empty.

ST_ExteriorShell returns a ST_MultiSurface value representing the outer shell of the solid.[1] This method is not applicable to ST_MultiSolid and its derived types.

ST_InteriorShells returns an array of ST_MultiSurface values representing the inner shells of the solid, i. e. the holes inside the object. For the extended standardized type hierarchy, this method is not applicable to ST_MultiSolid and its derived types. This method is not mandatory because the underlying DBMS may not support the feature T571, "Array-returning external SQL-invoked functions".

[1] Note that an adjustment to the definition of multi-surfaces is necessary if the boundary of a solid or multi-solid is to be described by that data type because multi-polygons must now allow common edges for the polygons in the collection.

ST_NumIntShells returns the number of inner shells of the solid. For the extended standardized type hierarchy, this method is not applicable to ST_MultiSolid and its derived types.

ST_InteriorShellN returns an ST_MultiSurface value representing the specified element in the array of inner shells of the solid. For the extended standardized type hierarchy, this method is not applicable to ST_MultiSolid and its derived types.

ST_NumFacets returns the number of facets (polygons) that were used to define a value of type ST_Polyhedron.

Additionally, we define constructor methods for each instantiable type in order to generate new polyhedron or multi-polyhedron values. The constructors take an extended well-known text (WKT) or well-known binary (WKB) representation as described in Section 5.3 as input. Another overloaded constructor exists that receives an ST_MultiPolygon value describing all the polygons that make up the surface of the three-dimensional object.

Finally, one new method *ST_IsShell* is to be added to the type ST_MultiSurface. This method tests if the given multi-polygon describes a geometry that is or could be the exterior shell of a solid or multi-solid. To that end, it verifies that all conditions that we explained in Section 5.2.1 are met, i. e. that the shell fully encloses a 3D object.

A modification to the semantics of the already existing ST_MultiPolygon type needs to be adopted in order to allow multi-polygons as input for the constructors or to return a shell as multi-polygon value. Currently, the SQL/MM spatial standard mandates that the parts of a multi-surface shall not spatially intersect and that the boundaries of any two pair-wise distinct surfaces in the ST_MultiSurface value shall at most intersect at a finite number of points. The first condition can be upheld in $\mathbb{R}^3$, but the second effectively limits the intersection of the boundaries to points or multi-points. For example, that makes it impossible that two polygons may share a common segment at the border but that is mandatory for the facets on the surface of a polyhedron.

Adapting existing Methods

The majority of the methods is defined for the root of the type hierarchy, ST_Geometry. Those methods need to be able to process 3D data as well. The following three aspects need to be addressed to reach a complete 3D solution:

1. The methods, e. g. *ST_NumPoints* inherited from supertypes for solids and multi-solids, i. e. ST_Geometry and ST_GeomCollection have to be implemented for 3D objects.

2. Any geometry (point, curve, surface, or solid) is to be handled in three-dimensions space.

3. Spatially comparing solids and multi-solids with non-three-dimensional geometries is necessary.

Implementing methods inherited from ST_Geometry or ST_GeomCollection can be accomplished in a traditional object-oriented fashion by overriding these methods for the new types. For example, the method *ST_NumPoints* can have its specialized implementation that knows how to determine the requested number for supported 3D objects if that becomes necessary.

The second issue concerns the dimensionality of the space in which the geometries are modeled. Existing products available today may allow the storage of a third (and even fourth) dimension, but all calculations are performed in 2D on the X/Y-plane [Wes04]. The code implementing this logic needs to be modified if, for example, the distance between two points is to be calculated in $\mathbb{R}^3$. Without such a modification, no usable 3D enhancement is possible and the closure of the spatial operations with respect to the underlying SRS would be lost. Another example is the intersection of two polyhedra, which could yield a point and linestring as result. Such a resulting geometry needs to be represented in 3D as well as value of type ST_GeomCollection. However, it is not necessary and also not a viable option to completely drop the existing support for two-dimensional data. Many applications require only two dimensions, and imposing the more complex algorithms for 3D calculations as a default would be counter-productive. The distinction whether a spatial operation should be performed in 2D or 3D space can be based on the SRS of the data, i. e. if the geometry is in $\mathbb{R}^3$, then the calculations are performed in a three-dimensional data space; otherwise, the already existing and usually better-performing 2D logic is exploited.

The current working draft of the SQL/MM spatial standard already defines the facilities to represent points, lines, and polygons as well as collections thereof in the three-dimensional space. The necessary properties of the geometries are specified. In particular, special considerations were added that all simple surfaces of type ST_Surface in 3D space shall be isomorphic to planar surfaces, and a linestring is only closed if the X, Y, and Z coordinates of the first and last point coincide. Some of the provisions in the working draft are actually not quite fitting for three-dimensional objects, for example the type ST_Surface shall be used to represent polyhedral surfaces, but it can't be used to describe the surface of a polyhedron as it does not allow for multiple single, planar surfaces. The type ST_MultiSurface has to be used for such a task instead.

The third issue in the listing above is automatically solved. Once any geometry can be created in $\mathbb{R}^3$, the overridden methods that take two geometries as input parameters like *ST_Intersects* can be implemented in such a way that comparisons and calculations between solids and non-solids (or collections of both) are dealt with properly.

5.3 Handling External Data Formats

Supporting additional complex data types in a relational database management system (RDBMS) requires that values of those data types can be constructed and stored in tables as well as retrieved by an application program, for example for visualization purposes. As we already stressed in Section 3.2, an external data format is mandatory for the data transport. Non-withstanding that multi-polygons can be used as a means to construct a polyhedron or multi-polyhedron, this section describes an enhancement to the standardized external data formats. The new representations are needed in order to preserve the type information in the respective format. Only the additions for the polyhedron and multi-polyhedron types are discussed. The current working draft of the SQL/MM spatial standard [ISO05a], which is not yet officially adopted, already contains the facilities to manage a third and even fourth dimension – Z and M coordinates – as part of points, curves, surfaces, and geometry collections. This situation has been taken into consideration and the respective elements are used.

We describe an extension to the well-known text and well-known binary representation in the following Sections 5.3.1 and 5.3.2, respectively. Our approach gives the encoding for polyhedra based on the existing definitions for polygons by grouping polygons to shells and shells to polyhedra. Multi-polyhedra are specified based on polyhedra. Another approach is presented in Section 5.3.3 where the properties of polyhedra are exploited for a more efficient and compact format. We adopt an index-based definition for the points, similar to the one that is used by the Computational Geometry Algorithms Library [CGA04]. All syntax definitions in the extended Backus-Naur Form (BNF). The full specifications can also be found in Appendix B.

5.3.1 Extending the WKT Representation

The well-known text (WKT) representation is an external data format used to encode geometries in a textual format. The *ST_AsText* method can be used to convert a geometry stored in a relational database to that format.

WKT comes with a rather simple structure. Each geometry is defined by a set of points, and parenthesis are used to group those points to lines or rings, polygons and multi-part geometries. The coordinates of different points and also the groups are separated by commas. A simple example for a multi-linestring with Z coordinates and two independent linestrings looks like the following. Each coordinate triple identifies a single point. The first value is the X coordinate, the second the Y coordinate and the last is the Z coordinate.

MULTILINESTRING Z((10 10 3, 20 20 8),(20 10 23, 20 15 17))

Listing 5.1: An example for the WKT representation

Listing 5.2 shows the syntax diagram for the extension of the WKT representation of polyhedra. The extended BNF notation in used for the definition. Listing 5.2 allows only X, Y, and Z coordinates for polyhedra. The specification for polyhedra with an additional M coordinate is omitted here and can be found in Appendix B.1.

```
<polyhedron text representation> ::=
    POLYHEDRON <3d zm> <polyhedron text>

<polyhedron text> ::=
      EMPTY
    | <polyhedron text body>

<polyhedron text body> ::=
    <left paren> <shell text>
        { <comma> <shell text> }... <right paren>

<shell text> ::=
    <left paren> <facet text>
        { <comma> <facet text> }... <right paren>

<facet text> ::=
    <left paren> <ring text>
        { <comma> <ring text> }... <right paren>

<ring text> ::=
    <left paren> <pointz text>
        { <comma> <pointz text> }... <right paren>

<pointz text> ::= <x> <y> <z> [ <m> ]

<3d zm> ::= Z | ZM

<x> ::= <number>
<y> ::= <number>
<z> ::= <number>
<m> ::= <number>
```

Listing 5.2: Extension of the WKT representation for polyhedra

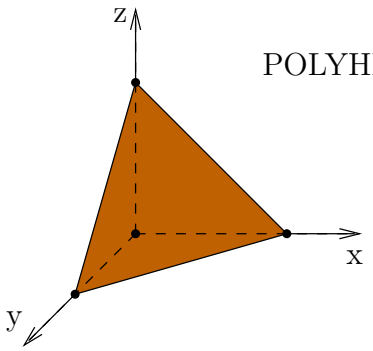
The keyword EMPTY is intended for "empty polyhedra". Such geometries may be the result of an intersection operation when the given input geometries do not intersect, for instance. Using the modified type hierarchy as in Figure 5.8 voids the need for that construct because empty geometries are adequately handled with a dedicated type.

A polyhedron is comprised of a number of shells (<*shell text*>). The first shell is always the exterior shell, and all (optional) subsequent ones are interior shells. Each shell is a non-empty multi-polygon, combining multiple facets. A facet is the same as a polygon and it is defined using rings. The different terminology is used to distinguish its role in the shell of polyhedra. Each ring consists of a number of point. The first ring of a facet is the exterior ring, and any additional ring represents a hole. All these structural information are indeed required to have an unambiguous representation of the object. For example, just allowing a set of rings to define the shell of a polyhedron and omitting the intermediate step of the facets would ignore the possibility of holes in facets. In other words, a polyhedron is specified via its boundary. That is also known as the *boundary representation* in computational geometry [Kar89].

Ideally, the BNF in Listing 5.2 would refer to the definition of the WKT format in the SQL/MM spatial standard and use the <*polygon text*> as non-terminal symbol to define shells. However, the SQL/MM spatial standard does allow an empty polygon as part of a multi-polygon. That discrepancy appears to be a problem in the standard, which may not have been intentional because a multi-polygon may be comprised of one ore more empty polygons. Allowing an empty shell or empty parts of a shell is not an option because it could lead to the situation that there is no exterior shell at all, giving an invalid polyhedron. Hence, the BNF is fully specified to the symbol <*number*>. This symbol is covered by the SQL/MM spatial standard and can be used as defined there.

Additional semantical conditions apply to polyhedra; those conditions are not expressed in the BNF. The shells must be closed, and the interior of the holes defined by each interior shell may touch at a finite number of points or linestrings but the shells shall not intersect. These semantical constraints cannot be expressed in BNF at all. Each shell has to consist of at least four facets. A ring of a facet requires at least four points (triangle) where the first and last point of a ring coincide. This condition could be specified by a more verbose BNF, but it does not add any real value.

An example for a polyhedron and its WKT representation is shown in *Figure 5.9*. The tetrahedron with a single shell is defined. A tetrahedron has four sides, i. e. four facets and each facet is actually a triangle. That is a minimal polyhedron.




```
POLYHEDRON Z ((((0 0 0, 1 0 0, 0 0 1, 0 0 0)),
               ((1 0 0, 0 1 0, 0 0 1, 1 0 0)),
               ((0 0 1, 0 1 0, 0 0 0, 0 0 1)),
               ((0 0 0, 0 1 0, 1 0 0, 0 0 0))))
```

Figure 5.9: WKT for a sample tetrahedron

The BNF for multi-polyhedra uses the previous definitions to introduce the additional grouping as *Listing 5.3* shows. Again, the keyword EMPTY is present for empty multi-polyhedra because no dedicated data type is used for that kind of geometries. The remaining definition only introduces another level of parenthesis that enclose the single polyhedra of the collection.

```
<multipolyhedron text representation> ::=
    MULTIPOLYHEDRON <3d zm> <multipolyhedron text>

<multipolyhedron text> ::=
      EMPTY
    | <left paren> <polyhedron text body>
          { <comma> <polyhedron text body> }... <right paren>
```

Listing 5.3: Extension of the WKT representation for multi-polyhedra

5.3.2 Extending the WKB Representation

The well-known binary (WKB) representation as specified in [ISO03d] and [ISO05a] is another external data format. As opposed to WKT, it encodes the geometry into a binary stream, avoiding rounding issues with the conversion from the internal binary to the external textual representation. The standardized method *ST_AsBinary* provides the mechanism for this conversion for all values of type ST_Geometry.

WKB is similar to the WKT representation, but the grouping of the points, rings or facets is achieved by giving the number of those structures in the header of each group. Another difference is attributed to the handling of multi-byte data values of various hardware platforms. The WKB representation includes – as its very first byte – the byte order (endianess) of the geometry encoding. The endianess is either *little endian* or *big endian* [Coh81].

Listing 5.4 shows the definition for the WKB representation of polyhedra with X, Y, and Z coordinates. The case where a geometry has M coordinates and, therefore, comes with an additional fourth dimension is not included. Instead, the full definition of the WKB can be found in Appendix B.2.

The format is close to the already standardized WKB representation of other geometry types, e. g. points, linestrings, and polygons. The BNF in Listing 5.4 shows the non-terminal symbol to *<pointz binary>*. That is necessary because the version of the WKB format in the current working draft of the SQL/MM spatial standard is not entirely applicable. For example, the ring of a polygon uses the symbol *<wkbringz binary>* in the standard. This symbol resolves to any one of the symbols *<linestringz binary>*, *<circularstringz binary>*, or *<compoundcurvez binary>*. Circular as well as compound curves are not permissible for facets of polyhedra. Additionally, *<linestringz binary>*

```
<well−knownz binary representation> ::=
      <pointz binary representation>
    | <curvez binary representation>
    | <surfacez binary representation>
    | <solidz binary representation>
    | <collectionz  binary representation>

<solidz binary representation> ::=
    <polyhedronz binary representation>

<polyhedronz binary representation> ::=
    <byte order> <wkbpolyhedronz>
        [ <num> <shellz binary>... ]

<shellz binary> ::= <num> <facetz binary>...
<facetz binary> ::= <num> <ringz binary>...
<ringz binary> ::= <num> <pointz binary>...
```

Listing 5.4: Extension of the WKB representation for polyhedra

elements have their own endianess and type identifier. Both are not required because the polyhedron itself already carries this information.

Geometry collections are a combination of several geometric primitives of the respective type. In particular, the complete WKB encoding of the primitive type is used including the byte order and the type identifier. Thus, the WKB representation for multi-polyhedra is defined as shown in *Listing 5.5*. The definition refers directly to the symbol *<polyhedronz binary representation>*.

```
<collectionz  binary representation> ::=
      <multipointz binary representation>
    | <multicurvez binary representation>
    | <multisurfacez binary representation>
    | <multisolidz binary representation>
    | <geometrycollectionz binary representation>

<multisolidz binary representation> ::=
    <multipolyhedronz binary representation>

<multipolyhedronz binary representation> ::=
    <byte order> <wkbmultipolyhedronz>
        [ <num> <polyhedronz binary representation> ]
```

Listing 5.5: Extension of the WKB representation for multi-polyhedra

The WKB format includes a numerical value to identify the type of the encoded geometry. It follows right behind the byte order. Those type identifiers are mandated by the SQL/MM spatial standard because they are essential for interoperability of the WKB encoding. *Table 5.1* shows the type identifiers for the four new data types. The chosen type identifiers align with the already existing identifiers. The standard uses type identifiers starting with 3,000,000 to denote geometries that contain Z coordinates. Type identifiers starting with 2,000,000 are used for types with Z as well as M coordinates. Polyhedra do need Z coordinates, so only these two groups need to be considered.

Geometry	Type id
<*wkbpolyhedronz*>	3,000,013
<*wkbmultipolyhedronz*>	3,000,014
<*wkbpolyhedronzm*>	2,000,013
<*wkbmultipolyhedronzm*>	2,000,014

Table 5.1: WKB type identifiers for polyhedra and multi-polyhedra

By means of an example, the WKB encoding of the polyhedron in Figure 5.9 is illustrated in *Figure 5.10*. The first line contains the byte order, the type id, the number of shells, and the number of facets in the first (and only) shell. Following those 13 bytes comes the information of each facet in a separate line.

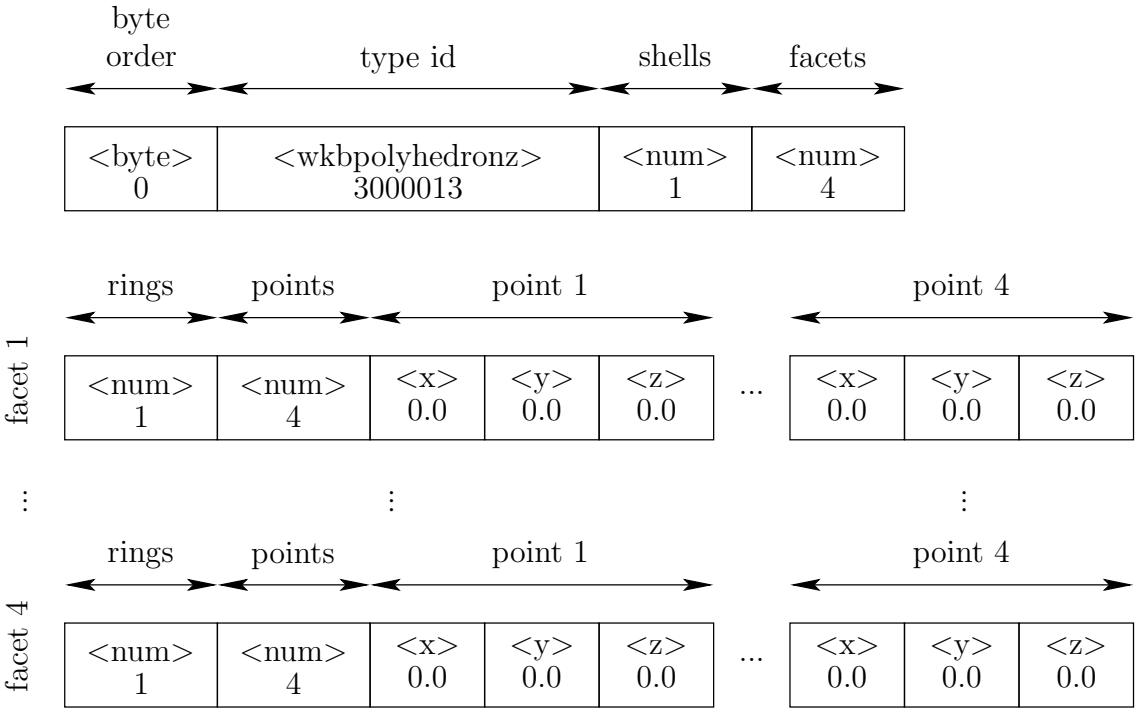


Figure 5.10: WKB for a sample tetrahedron

5.3.3 Index-Based Representations

We defined the means to extend the WKT and WKB representations from [ISO05a] in the previous two Sections 5.3.1 and 5.3.2. The extension is closely aligned with the already existing specifications. Both formats are not very efficient with respect to space consumption and robustness for higher-dimensional geometries. One property of polyhedra is that each non-trivial point in a shell belongs to at least three facets. That leads to a three-fold space consumption for each point. Given the inherent complex nature of polyhedra, it has a significant impact when such geometries are transferred between the database server and an application.

Another problem is the potential for inconsistencies if not exactly the same coordinates are used for the same point in all three (or more) facets. Rounding differences could lead to polyhedra that do not have closed shells if only one of the points is slightly off target, making the format not very robust.

The solution lies in the adoption of a structure that consists of two parts for polyhedra. First, all points are specified once in an array, i. e. without redundancies. The second part defines the actual shape of the polyhedron in the usual way. Instead of specifying the point coordinates (again) in the second part, the index of the point in the array is used. The approach is also known as *indexed face sets* [Kar89]. This idea is very similar to color lookup tables used in graphics file formats, e. g. GIF [BS95].

The WKT representation for polyhedra using such a format is shown in *Listing 5.6*. All not explicitly specified non-terminal symbols are defined as in Listing 5.4. As can be seen in the BNF diagram, the place of the first shell is now occupied by the array containing all the points of the polyhedron, and there are no points in rings of the single facets but rather a set of index keys, which refer to the array. The array is enclosed in parenthesis to follow the spirit of WKT even if that is not strictly necessary. Each index is an <*unsigned integer*>, which is defined in Subclause 5.3 of [ISO03i].

Both variations of the WKT representation can be distinguished based on the keyword INDEX. One issue arises if both versions shall be supported and that is the result of the method *ST_AsText* because only one of the two formats can be produced. Either the method is parameterized or two methods are needed for all spatial data types. Using both formats complicates the interface unnecessarily so that the recommendation is that the SQL/MM spatial standard should only adopt one format, the index-based one. If the standard only defines the index-based version shown here, the keyword INDEX can be omitted because the type name POLYHEDRON is already decisive enough.

The definition of the WKT for multi-polyhedra can be used unmodified from Listing 5.3 and it is not repeated here. However, the INDEX keyword is added for consistency reasons. It marks the deviation from WKT as it is used for the representation of points, linestrings, and polygons.

For completeness, the WKB format is handled in the same fashion as the WKT representation and *Listing 5.7* shows the BNF syntax. Besides the initial array containing

```
<polyhedronz index text representation> ::=
    POLYHEDRON <3d zm> INDEX <polyhedron index text>

<polyhedron index text> ::=
      EMPTY
    | <polyhedron index text body>

<polyhedron index text body> ::=
    <left paren> <point index list> <shell index text>
        { <comma> <shell index text> }... <right paren>

<point index list > ::=
    <left paren> <pointz text>
        { <comma> <pointz text> }... <right paren>

<shell index text > ::=
    <left paren> <facet index text>
        { <comma> <facet index text> }... <right paren>

<facet index text > ::=
    <left paren> <ring index text>
        { <comma> <ring index text> }... <right paren>

<ring index text> ::=
    <left paren> <index>
        { <comma> <index> }... <right paren>

<index> ::= <unsigned integer>
```

Listing 5.6: Index-based WKT for polyhedra

the points, the only difference when compared with the WKB given in Listing 5.4 can be found in the definition of the rings, which do not specify the points explicitly but rather a number of index keys that refer to the array.

```
<polyhedronz index binary representation> ::=
    <byte order> <wkbpolyhedronindexz>
        [ <num> <wkbpointz binary>...
          <num> <shellz index binary> ]

<shellz index binary> ::= <num> <facetz index binary>...
<facetz index binary> ::= <num> <ringz index binary>...
<ringz index binary> ::= <num> <index>...

<polyhedronzm index binary representation> ::=
    <byte order> <wkbpolyhedronindexzm>
        [ <num> <wkbpointzm binary>...
          <num> <shellzm index binary> ]

<shellzm index binary> ::= <num> <facetzm index binary>...
<facetzm index binary> ::= <num> <ringzm index binary>...
<ringzm index binary> ::= <num> <index>...

<index> ::= <uint32>
```

Listing 5.7: Index-based WKB for polyhedra

The index-based WKB format given here uses intentionally the non-terminal symbol *<wkbpolyhedronindexz>* and not *<wkbpolyhedronz>* to represent the type identifier. As with the WKT representation, it is possible that an implementation of the SQL/MM spatial standard could support both versions of the WKB. A different type identifier would be the means to tell both versions apart. *Table 5.2* suggests type identifiers to be used for the index-based well-known binary representation of polyhedra. No new type identifiers are necessary for multi-polyhedra. Multi-polyhedra are defined by a number of polyhedra, and each polyhedron is given by its full WKB representation, a including the specific type identifier.

Geometry	Type id
<wkbpolyhedronz>	3,000,015
<wkbpolyhedronzm>	2,000,015

Table 5.2: Type identifiers for polyhedra in index-based WKB

The WKB representation allows for an exact evaluation of the space requirements for a polyhedron by comparing both versions of the format. The index-based format requires

at least $3 \cdot 8 + 3 \cdot 4 = 36$ bytes per point. Each point consists of three <*double*> values, each 8 bytes long, and each point occurs at least three times in the polyhedron definition, requiring 4 bytes for each index key. The first version of the format takes twice the amount $3 \cdot 3 \cdot 8 = 72$ bytes because each point itself occurs three times and a point has at least 24 bytes due to its three coordinates. The gap grows even further if a point exists in more than three facets of a shell or if M coordinates are used additionally.

5.4 SQL/MM Spatial Information Schema

The spatial information schema that we initially introduced in Section 2.3.5 contains the meta data that is used to describe the spatial data available in the database. The meta data includes the view ST_GEOMETRY_COLUMNS that shows all spatial columns, i. e. the columns of any table that has a declared type of ST_Geometry or any of its proper subtypes. The view ST_SPATIAL_REFERENCE_SYSTEMS lists all currently defined spatial reference system (SRS) and the view ST_UNITS_OF_MEASURE defines all linear and angular units that can be used to measure lengths or areas. The fourth and final view in the information schema ST_SIZINGS is usually not directly needed by an application as it only provides a means to communicate the settings for the implementation-defined variables.

Analyzing each view with respect to the addition of data types for 3D objects reveals that ST_GEOMETRY_COLUMNS is not affected by 3D support at all. The new data types ST_Solid and ST_MultiSolid are subtypes of ST_Geometry and the recursive definition of the view will automatically identify any column with the new data types as spatial column.

It was stated that the addition of three-dimensional data to the SQL/MM spatial standard should allow the coexistence of data in $\mathbb{R}^2$ and $\mathbb{R}^3$. Thus, the definitions of two-dimensional SRS as well as three-dimensional ones will have to be accessible through the view ST_SPATIAL_REFERENCE_SYSTEMS. For each SRS, the view shows the identifying name, the numeric identifier, the organization that defined the system along with the identifier assigned by that organization, the actual definition and a general description for it. The very same information can be provided for three-dimensional SRS. However, a distinction between 2D and 3D can then only be based reliably on the definition itself. Therefore, it is proposed to add another attribute to the view which will reflect the dimensionality of the data space used for the spatial calculations, i. e. 2 for $\mathbb{R}^2$ and 3 for $\mathbb{R}^3$. Thus, the new relational schema for the view will be as shown in *Listing 5.8*.

```
ST_SPATIAL_REFERENCE_SYSTEMS ( srs_name, srs_id,
   organization, organization_coordsys_id,
   definition, dimensionality, description )
```

Listing 5.8: Extended schema for ST_SPATIAL_REFERENCE_SYSTEMS

The modified view is based on a table with the same name in the spatial definition schema as defined in the standard. The new attribute can be directly added to that table using the data type INTEGER and disallowing NULL. A check constraint has to be added to restrict the values for the attribute. The modifications can be applied by altering the table as *Listing 5.9* demonstrates. *ST_MaxDimensionality* denotes a new implementation-defined variable that describes the maximum dimensionality supported for spatial operations. The value for such variables are listed in the ST_SIZINGS view of the information schema.

```
ALTER TABLE st_spatial_reference_systems
   ADD COLUMN dimensionality NOT NULL,
   ADD CONSTRAINT dimensionality_value
      CHECK ( dimensionality BETWEEN 2 AND
                  ST_MaxDimensionality )
```

Listing 5.9: Extending the Spatial Definition Schema

The definition of ST_UNITS_OF_MEASURE does not have to be adjusted. The units used for two-dimensional SRS are also applicable for three-dimensional ones. However, it may be possible that an implementation of the standard provides additional units, for example new units like "light year" for astronomical applications [Sch06], which can now be supported more adequately with 3D operations.

The addition of a new attribute to table ST_SPATIAL_REFERENCE_SYSTEMS in the definition schema as shown leads to the declaration of a new implementation-defined meta-variable. All such meta-variables are listed in view ST_SIZINGS. The addition of a new variable does not have an impact on the definition of the view itself and the view remains adequate with three-dimensional data types being supported. It should be noted that a product implementing the standard may have to revisit its specifications in the view. For example it has to be verified whether the maximum length of the WKT representation of a geometry, the variable *ST_MaxGeometryAsText*, has to be increased given the higher complexity and size for the definitions of solids and multi-solids.

5.5 The Spatial 3D Extender for DB2

The previous sections introduced the concepts and definitions to incorporate three-dimensional objects into the SQL/MM spatial standard [ISO03d], using the current working draft of the standard as a base line. The DB2 Spatial Extender [IBM04d] is an actual and conforming implementation of this standard. However, the DB2 Spatial Extender is only manages two-dimensional data. It already provides some mechanisms to store Z and M coordinates in the geometries, even though there are only very few ways to exploit such information, and no support for operations in 3D space is available.

This section describes a proof-of-concept implementation of a 3D Extender for the solid and multi-solid types to represent three-dimensional objects, especially polyhedron and collections of polyhedra. Based on the work presented in [Bit05, SB06], the DB2 Spatial Extender was used as the product of choice and its SQL interface, i. e. the type hierarchy and the spatial routines, were adjusted for the additional functionality.

The Computational Geometry Algorithms Library (CGAL) [CGA04] was chosen for the spatial calculations in $\mathbb{R}^3$. Other open-source libraries for 3D operations, for example LEDA [MN99], are able to handle tetrahedra, which are the basic building blocks for more complex three-dimensional objects. However, CGAL comes with a distinct advantage of supporting arbitrary polyhedra and even simple points, lines, and polygons in $\mathbb{R}^3$. Thus, the implementation effort for the 3D Extender became containable for our implementation.

We introduce the CGAL library and its relevant components in Section 5.5.1. We add new data types to handle solids directly to the Spatial Extender's type hierarchy. Section 5.5.2 describes the details how the new types are be integrated. It includes an explanation of the relevant attributes of the types and how the data of the three-dimensional objects is stored by the 3D Extender. Based on that preparation of the type hierarchy, we go into the details of adding new methods or modifying a selected set of existing spatial methods in Section 5.5.3. The methods *ST_Buffer* and *ST_Intersection* can now deal with polyhedra, for example. The issue of indexing three-dimensional objects is briefly raised in Section 5.5.4, even though this topic is irrelevant to the SQL/MM spatial standard. We analyze the performance of the integration in Section 5.5.5 and compared it to the results from Bittner [Bit05]. The efficiency is measured for various scenarios and theoretical considerations complete the analysis. Finally, we implement a visualizer to be able to verify the correctness of the results computed by the 3D operations. Section 5.5.6 explains the features of this tool. We summarize the findings from the implementation of the prototype in Section 5.5.7.

5.5.1 Overview of CGAL

The ability for spatial calculations in $\mathbb{R}^3$ is crucial to support fully three-dimensional objects in the DB2 Spatial Extender. Operations in three-dimensional space have generally a higher complexity than in $\mathbb{R}^2$ due to the additional dimension [HL93]. However, the primary focus of the work presented here are integration aspects for the spatial type hierarchy in a relational database and not the implementation of specific algorithms for calculations in $\mathbb{R}^3$. Thus, the existing library Computational Geometry Algorithms Library (CGAL) [CGA04] was employed to perform the 3D calculations despite its minor shortcomings that we briefly mentioned in Section 5.5.7.

The development of the library was initiated by a consortium of several universities and institutions, for instance ETH Zürich, FU Berlin, Max Planck Institute and others.

CGAL became an open source project in 2003 and can now be used, modified and adapted freely. It may not have the optimal performance, but the supported functionality and the openness of the source code were essential to the prototypical implementation of the 3D Extender.

CGAL was implemented using the C++ programming language [Str97]. The library is not directly tailored to manage geometries. Instead, it implements a topological model and conversions between the geometries and their topological representation become necessary [OSQZ02]. CGAL consists of three main components:

Core This component is the kernel and it defines a set of basic geometric types and operations on those types. These basic types are points, vectors, lines, triangles and tetrahedra, and they are the building blocks for more complex geometries like polygons or polyhedra. Every basic type comes with its specific set of functions, for example the area of a triangle or length of a line can be gathered. Additionally, operations like distance and intersection calculation on those basic geometric types are possible.

Basic Library This part of CGAL uses the Core to build and represent more complex geometries, including polyhedra and Nef-polyhedra [Nef78]. Operations on the complex types are supported as well, for example triangulation or computation of the convex hull. Internally, the implementation of those operations is heavily built on the logic in Core by breaking down the complex geometries to basic geometries.

Support Layer Graphic presentation and the logic for different numeric types with higher precision can be found here. Although this part of CGAL is implicitly used by the 3D Extender, it is not a central part for geometric operations.

We used only a selected set of all data structures and functions available in CGAL for the extension of the DB2 Spatial Extender. In particular, only the structures for polyhedra and the more general Nef-polyhedra are relevant. Both data types shall be introduced briefly.

CGAL Polyhedra

The C++ class *Polyhedron_3* is provided to represent and work with polyhedra. Internally, a data structure based on half-edges[2] is used to define a polyhedron. That data structure is similar to the doubly connected edge list (DCEL) [BKOS00]. The existing points, half-edges and areas are stored together with additional information about the respective neighbors, i. e. adjacent points, twin half-edges and others. A more detailed introduction can be found in [Ket99].

[2] An undirected edge can be represented by two directed half-edges, each facing in the opposite direction of the other.

The polyhedron class does not offer any set-based functionality because polyhedra are not closed with respect to spatial operations like intersection or difference. For example, the intersection of two (touching) polyhedra could be a single point, which is not a polyhedron itself. The main advantage of the class and its primary usage as part of the presented extension can be found in the evaluation routines that take a given polyhedron description and verify that the description actually represents a valid polyhedron, i. e. that all polygons of a shell are planar and that a shell is closed. That can be used for validation purposes in the constructor functions that generate a new ST_Polyhedron or ST_MultiPolyhedron value from the WKT or WKB representation.

CGAL Nef-Polyhedra

A CGAL polyhedron can be converted to a CGAL Nef-polyhedron. The Nef-polyhedron [Nef78] is the data structures that does not carry the mentioned shortcomings due to a broader scope of the geometries that can be represented. Points, linestrings, polygons, polyhedra and homogenous and heterogenous collections of those types can be described by Nef-polyhedra. The class *Nef_polyhedron_3* implements the data structure for three-dimensional Nef-polyhedra. Curved segments or curved surfaces are approximated by linear or planar objects, respectively.

Definition 3 (Nef-polyhedron)
A Nef-polyhedron of dimension d is a set of points $P \subseteq \mathbb{R}^d$ that is constructed by a finite number of half-spaces using the complement and intersection operations only.

All set operations can be reduced to complement $\complement$ and intersection $\cap$ operations. For example, the union $\cup$ of two polyhedra P_1 and P_2 is the complement of the intersection of the complements of both polyhedra, i. e. $P_1 \cup P_2 = \complement(\complement(P_1) \cap \complement(P_2))$. All set operations performed on Nef-polyhedra result again in a new Nef-polyhedron. Thus, Nef-polyhedra are closed regarding set operations. The mathematical and geometric properties [GHH+03, CGA04] of Nef-polyhedra are versatile, but only of marginal interest for the 3D Extender.

CGAL polyhedra can be converted directly to CGAL Nef-polyhedra. Unfortunately, the construction of Nef-polyhedra from points, linestrings and polygons is not directly supported. Either half-spaces have to be used for the construction according to the definition of Nef-polyhedra, or the CGAL-internal representation is to be put to work. Given that the prototype has to deal with those geometries in 3D space as well, new constructor functions were needed. Otherwise, comparison operations between polyhedra and other types of geometries could not be provided.

The internal storage structure of CGAL Nef-polyhedra is rather complex. It contains information about points, edges, 2D and 3D objects, all of which are connected amongst each other. The most important elements will be briefly introduced.

edge An (undirected) edge in a Nef-polyhedron is implemented by two directed and linked edges, so-called *half-edges*. The half-edges of such a pair have the opposite direction.

facet A facet is an area represented by directed cycles of its (half-)edges. The half-edges in a cycle are only linked in one direction. The cycle of an opposite facet also has the opposite direction.

shell A shell is the surface of a three-dimensional object. A shell is defined by facets, and through facet by edges and points.

volume A volume is actually a three-dimensional object. It consists of multiple shells, one outer and zero or more inner shells, which define the boundary of the volume. The inner shells define holes inside the object.

The operations on a CGAL Nef-polyhedron cover the full spectrum of set operations like intersection, union, complement and difference. The result of such an operation is again a Nef-polyhedron. Thus, it is not only necessary to construct a Nef-polyhedron from external data formats like WKT and WKB or the representation used internally in the new 3D Extender data types ST_Polyhedron and ST_MultiPolyhedron but also to convert Nef-polyhedra to the representation used by the 3D Extender. Converting the spatial data between the topological and the geometric representation was already described in [OSQZ02] as a method to bring these two worlds together. The encoding for the information of the three-dimensional objects in the SQL data types is described in the next section as it directly relates to the spatial type hierarchy.

5.5.2 Implementation of the Extended Type Hierarchy

This section describes how the new data types to represent three-dimensional objects are added to the type hierarchy of the DB2 Spatial Extender. The extender employs structured types [Gep02] to model the spatial hierarchy. Structured types are basic constructs of the object-relational functionality offered by DB2. It is possibly to define a type hierarchy, where a subtype inherits attributes and methods from its supertype. Furthermore, structured types can be nested, i. e. a structured type may have attributes that are themselves structured. Next to distinct types, structured types are user-defined type (UDT).

The DB2 system catalog views provide all the information that is necessary to figure out the exact definition of the spatial data types. The view SYSCAT.ATTRIBUTES reveals the attributes and their data types of the type ST_Geometry. Any new types can be directly added to the type hierarchy with this information available and the existing type structure is fully taken advantage of. *Listing 5.10* shows the SQL statements used to add the four new types ST_Solid, ST_Polyhedron, ST_MultiSolid, and ST_MultiPolyhedron.

Following the naming conventions of the extender, all types are placed in the schema db2gse. The statements realize the new types exactly as was shown in Figure 5.6. The types ST_Solid and ST_MultiSolid are not instantiable, i. e. abstract types. The other two types are used to represent arbitrary polyhedra and, thus, both types must be defined as instantiable.

```
CREATE TYPE db2gse.ST_Solid UNDER db2gse.ST_Geometry
   NOT INSTANTIABLE
   WITHOUT COMPARISONS
   NOT FINAL
   MODE DB2SQL;

CREATE TYPE db2gse.ST_Polyhedron UNDER db2gse.ST_Solid
   INSTANTIABLE
   WITHOUT COMPARISONS
   NOT FINAL
   MODE DB2SQL;

CREATE TYPE db2gse.ST_MultiSolid
   UNDER db2gse.ST_GeomCollection
   NOT INSTANTIABLE
   WITHOUT COMPARISONS
   NOT FINAL
   MODE DB2SQL;

CREATE TYPE db2gse.ST_MultiPolyhedron
   UNDER db2gse.ST_MultiSolid
   INSTANTIABLE
   WITHOUT COMPARISONS
   NOT FINAL
   MODE DB2SQL;
```

Listing 5.10: Definition of 3D related data types in DB2

The new types do not add any attributes – none of the direct or indirect subtypes of ST_Geometry does. All attributes that store the geometry information exist in the root type of the hierarchy. The reason for such an implementation can be found in the mechanisms to pass structured values from the database engine to external code that implements a certain function or method via transform functions (like CGAL is accessed with user-defined functions (UDFs)). Transform functions – as defined by the SQL standard and implemented by DB2 [CCN+99] – are the means to serialize a structured type so that it can be communicated to the external code. A transform function has to be defined for a type and applies to that type and all its subtypes. If a subtype adds

new attributes, a new transform function is needed. Although it is possible to do that, the bottom line would be that each external function or method has to be implemented in several different variations, once for each combination of possible input values. That leads to a tremendous increase of complexity of the extender. In order to avoid this issue, the transform functions of the DB2 Spatial Extender are written in such a way that they serialize all the spatial types in the same way, regardless of the specific type. It requires the same (or at least a very similar) definition of all the types, and placing all attributes in ST_Geometry and not adding new attribute in the subtypes is the practical consequence. The very same considerations apply to the new types for 3D objects.

Storing the Geometry Data

The types for solids and multi-solids inherit the attribute points from ST_Geometry. This attribute is of type BLOB and can store binary data up to a length of 1 MB. The size restriction automatically applies to the internal storage scheme of solids as well.

The DB2 Spatial Extender uses an internal format to store the coordinate information of points, linestrings, polygons and geometry collections. The internal format is based on an integer representation of the coordinates, and compression is used to save space. The internal format is not documented in all details, so it is not possible to use the same format for polyhedra and multi-polyhedra. Thus, the 3D Extender has to choose its own encoding and cover any issues that arise from this approach. [Bit05] uses the well-known binary representation to encode the geometries in the binary stream. Given the size restriction of 1 MB and that a single point needs 24 bytes and occurs at least three times, a 3D object can have about 14500 points at most. Another result is that a majority of the time is lost to convert geometries in 3D space from the WKB format to CGAL Nef-polyhedra. The approach used in the 3D Extender [SB06] uses directly the representation of the Nef-polyhedra and applies a compression algorithm to reduce the space requirements. As we have shown in Section 5.5.5, the space requirements are still higher than with the WKB, so a productized version of the DB2 Spatial Extender with 3D support should lift the 1 MB restriction in the ST_Geometry type.

We base an alternative implementation on the already existing extensibility mechanisms in DB2 – without touching or modifying the DB2 Spatial Extender's data types. The data type ST_Geometry includes an attribute ext that has a declared type named SE_Extension. This data type is actually a structured type. The attribute is not used by any of the functions or methods of the DB2 Spatial Extender. The more complex description of three-dimensional geometries can be stored in a larger sized binary large object (BLOB) embedded in a new subtype of SE_Extension. Values of the type SE_Extension or its proper subtypes can be stored and accessed via the ext attribute. However, a disadvantage is the need to adjust the transform functions. To avoid this work, the 3D Extender sticks with the 1 MB BLOB only.

Handling the Internal Type Identifier

The DB2 Spatial Extender keeps an indication about the geometry type in the attribute geometry_type. This value is used in various functions and methods to determine if the current geometry is a point, linestring, or polygon. Additionally, the identifier encodes the information whether Z and/or M coordinates are present in addition to the X and Y coordinates. Strictly spoken, this information is redundant and would not be necessary, but it helps to improve performance in some situations.

Proper type identifiers need to be assigned to the new types of the Extender's type hierarchy. Some type identifiers are already defined for the existing types. The actual values and their meaning can be derived from the Spatial Extender's transform functions. Additionally, methods like *ST_Is3D* and *ST_IsMeasured* reveal that the two least significant bits encode the presence of Z and M coordinates and the modulo function is called to filter out those two bits. If bit 0 is set, the geometry has M coordinates; otherwise M coordinates are not used. Bit 1 stores the same information for Z coordinates, i. e. Z coordinates exist if the bit is set.

The new geometries for polyhedra and multi-polyhedra always contain Z coordinates. Thus, it is mandatory that the chosen type identifiers have bit 1 set. The bit 0 can vary, so that the two data types need four type identifiers. *Table 5.3* shows the type identifiers used for polyhedra and multi-polyhedra, depending on the presence of X, Y, Z, and M coordinates. The type identifiers are picked in such a way that all functions or methods of the DB2 Spatial Extender that evaluate the type identifiers for Z and M coordinates do not have to be modified, i. e. their modulo-operations return the desired result also for the new types. The type identifiers 48, 49, 52, and 53 are reserved but not used because they would represent polyhedra without Z coordinates.

Geometry type	Coordinates	Type Id
ST_Polyhedron	X, Y, Z	50
ST_Polyhedron	X, Y, Z, M	51
ST_MultiPolyhedron	X, Y, Z	54
ST_MultiPolyhedron	X, Y, Z, M	55

Table 5.3: DB2 type identifiers for polyhedra and multi-polyhedra

Settings for all other Attributes

The geometry type ST_Geometry has additional attributes besides the two that were already discussed. The values to be stored in those attributes are apparent and shall be summarized here:

srid is the numeric identifier of the SRS. This identifier needs to be in the range of 1,000,000,000 to 1,999,999,999 as this range is reserved for geometries in $\mathbb{R}^3$. Note that the range starting with 2,000,000,000 is already reserved by the DB2 Spatial Extender for geodetic coordinate systems.

numPoints stores the information how many points were used to define the geometry. The number of points in each ring is counted for (multi-)polyhedra.

xMin, xMax, yMin, yMax, zMin and zMax contain the coordinates of the minimum bounding box (MBB)[3] of the geometry.

mMin and mMax stores the minimum and maximum M coordinates found in the geometry if M coordinates are present.

area The sum of the area of all polygons in the shells of the (multi-)polyhedron is stored in this attribute for a faster access.

The attributes length and anno_text are not used by the 3D Extender. The original usage of the first attribute is for the length of linestrings and the length of the boundary of polygons, but it has no meaning for 3D objects. An annotation can be stored in the second attribute, but the DB2 Spatial Extender does not provide any documented interfaces to manage the annotation in the first place.

5.5.3 Adding and Modifying Spatial Functions and Methods

A representative set of spatial functions and methods is implemented in the 3D Extender. The functionality includes the constructors for the types ST_Polyhedron and ST_MultiPolyhedron as well as an enhancement to the factory function ST_Geometry for the new types. Furthermore, the conversion methods *ST_AsText* and *ST_AsBinary* are implemented to extract any geometry that was stored in the database or created by any spatial function. Generating a new 3D geometry via spatial set operations from two given 3D geometries is possible with the methods *ST_Buffer* and *ST_Intersection*. Comparing two three-dimensional objects whether they overlap, intersect or touch is a basic requirement for spatial joins. The DB2 Spatial Extender provides different functions for this purpose, each function for a specific spatial set operation. The 3D Extender redefines the functions *ST_Intersects*, *ST_Equals*, *ST_Contains* and *ST_Disjoint*. Finally, the methods *ST_BoundingArea* and *ST_Volume* are available from the group of routines that return properties of a geometry. This section explains the steps that were necessary to define all mentioned functions and methods.

[3] The *minimum bounding box* is the smallest cuboid that fully contains the geometry and whose sides are parallel to the axes of the coordinate system. Thus, it is an extension of the minimum bounding rectangle (MBR) that is applicable in two-dimensional coordinate systems.

The majority of the DB2 Spatial Extender routines are implemented as external UDFs using the C [KR88] and C++ [Str97] programming languages. The source code is proprietary and not available to add the described enhancements directly. Thus, another approach was taken. Each external function is registered in the database using the standardized routine name, e. g. *ST_Intersects*. Registering the external routine using a different function name and replacing the routine in the SQL database opens the opportunity to slip in logic that decides whether 2D or 3D calculations are required and then invokes the respective routine. *Figure 5.11* illustrates this scheme. The replacement of the original SQL routine is a wrapper that either calls the original 2D routine or the new 3D logic.

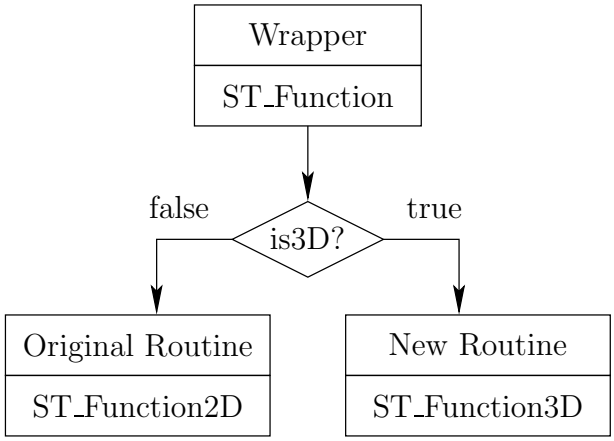


Figure 5.11: Adding logic to invoke 3D specific routines

The decision whether the 3D routine or the original 2D version is to be called is based on the spatial reference system (SRS) of the input geometry. We propose the addition of an attribute to the view ST_SPATIAL_REFERENCE_SYSTEMS in Section 5.4. Using the numeric identifier of the SRS associated with the geometry, this attribute can be queried. However, the 3D Extender adopted a simpler but not as flexible approach. The srid of the geometry is consulted and if the identifier of the SRS is in the interval $[1000000000, 2000000000)$ then the 3D specific version of the routine is invoked; otherwise, the original DB2 Spatial Extender implementation performs its job. [Bit05] tried to decide based on the presence of Z coordinates whether to call CGAL or the DB2 Spatial Extender logic. However, that would break existing applications that do want to store Z coordinates as additional information only and not perform calculations in $\mathbb{R}^3$. Therefore, the SRS-based approach is more suitable. *Listing 5.11* shows the SQL expression to decide whether the original DB2 Spatial Extender function is to be invoked or the new 3D specific one. The subselects in the `THEN` and `ELSE` branches are needed due to a particularity of DB2's implementation of `CASE` expressions. DB2 treats them like functions, implying that all parameters are evaluated before the `CASE` itself is considered.

Thus, the THEN and the ELSE branches are both evaluated regardless of the condition in the WHEN branch. If the external functions are called directly without being wrapped into the subselect, the original 2D version will return an error if the geometry (like a polyhedron) is not known to it. The subselects verify the respective condition again in their where-clauses, ensuring that the external code is only invoked when required.

```
CASE
   WHEN geometry..srid / 1000000000 = 1
   THEN ( SELECT GseName3D(geometry)
          FROM   sysibm.sysdummy1
          WHERE  geometry..srid / 1000000000 = 1 )
   ELSE ( SELECT GseName2D(geometry)
          FROM   sysibm.sysdummy1
          WHERE  geometry..srid / 1000000000 <> 1 )
END
```

Listing 5.11: Deciding between 2D and 3D logic

The described mechanism of inserting a CASE expression as an exit to the 3D functionality is only needed for original DB2 Spatial Extender routines that cannot be specialized for polyhedra and multi-polyhedra. For example, generating a buffer around a point in $\mathbb{R}^3$ makes the exit truly necessary. Methods specific to the 3D types only do not require this mechanism.

The different groups of spatial functions and their 3D enhancements are described subsequently. In each case it is explained if the 3D routines are invoked using the described wrapper or if another mechanism is more appropriate.

Transform Functions

DB2 Spatial Extender defines transform functions to pass geometry values between the SQL database and the external C++ code of the routines. The FROM SQL transform function accesses the attributes of the geometry, queries the Spatial Extender catalog table containing the exact definition of the spatial reference system, and passes the attributes and the SRS to the external code. Likewise, the TO SQL function takes the results produced by the external code, constructs a value of the respective subtype of ST_Geometry and populates its attributes with the parameters returned from the external function [Sto05b].

The transform function that constructs the spatial value in the SQL database needs to be rewritten so that it can handle polyhedra and multi-polyhedra if they are returned as result of an external function, for instance from a constructor function. The differentiation between the geometry types returned by the external code is done with the accompanying geomType value as shown in *Listing 5.12*.

```
CREATE FUNCTION db2gse.GeomFromUdf3D (
      srsId INTEGER, numPoints INTEGER, geomType SMALLINT,
      xMin DOUBLE, xMax DOUBLE, yMin DOUBLE, yMax DOUBLE,
      zMin DOUBLE, zMax DOUBLE, mMin DOUBLE, mMax DOUBLE,
      length DOUBLE, area DOUBLE, pointsData BLOB(1048576) )
   RETURNS db2gse.ST_Geometry
   LANGUAGE SQL  CONTAINS SQL
   DETERMINISTIC  NO EXTERNAL ACTION
   RETURN CASE
      WHEN geomType BETWEEN 4 AND 7
      THEN db2gse.ST_Point()..geometry_type(geomType)..
           srid(srsId).. numPoints(numPoints)..
           xMin(xMin)..xMax(xMax)..yMin(yMin)..yMax(yMax)..
           zMin(zMin)..zMax(zMax)..mMin(mMin)..mMax(mMax)..
           area(area)..points(pointsData)
      -- handle linestrings, polygons, and collections of
      -- points, linestrings, and polygons like points
      WHEN geomType BETWEEN 48 AND 51
      THEN db2gse.ST_Polyhedron()..geometry_type(geomType)..
           srid(srsId).. numPoints(numPoints)..
           xMin(xMin)..xMax(xMax)..yMin(yMin)..yMax(yMax)..
           zMin(zMin)..zMax(zMax)..mMin(mMin)..mMax(mMax)..
           area(area)..points(pointsData)
      -- handle ST_MultiPolyhedron like ST_Polyhedron
   END;

CREATE FUNCTION db2gse.GeomToUdf3D ( g db2gse.ST_Geometry )
   RETURNS ROW ( srsId INTEGER, pointsData BLOB(2M) )
   LANGUAGE SQL  CONTAINS SQL
   DETERMINISTIC  NO EXTERNAL ACTION
   RETURN VALUES ( g..srid, g..points );

CREATE TRANSFORM FOR db2gse.ST_Geometry TransformGroup3D (
   FROM SQL WITH FUNCTION db2gse.GeomToUdf3D,
   TO SQL WITH FUNCTION db2gse.GeomFromUdf3D );
```

Listing 5.12: Transforms for the 3D Extender

For the transform function that serializes a structured value to pass it to the external code, the UDF *GseGeomSRSToUDF* could be taken as it is implemented in DB2 Spatial Extender. Although the encoding of the points attribute uses originally an internal, not-documented encoding, this situation does not apply if all geometries created in $\mathbb{R}^3$ are handled by the extension of the 3D Extender. If the 3D Extender completely controls

the generation of all new geometries, it can choose its own encoding for the points attribute even for points, linestrings and polygons. The original `FROM SQL` transform function is more complex than it has to be because the 3D Extender does not need any information about the SRS besides the knowledge that the data is in 3D. However, it will become necessary to send additional information to the external code if multiple 3D coordinate systems shall be supported in the future. The 3D specific logic only uses a single coordinate system, so the issue does not arise. Listing 5.12 sketches the SQL code to create the necessary transform functions. The two transform functions are combined into a transform group named TransformGroup3D.

Once the transform group is defined we can used it when the 3D specific functions are registered in the database, for example in the constructors. These functions take automatically advantage of it for the necessary serialization of the geometry attributes.

Constructors

DB2 Spatial Extender provides, as mandated by the SQL/MM spatial standard, a set of constructor functions. Each instantiable type has its own constructor. Additionally, the function *ST_Geometry* exists as a factory function [GHJV95] that generates a value of the proper spatial subtype, depending on the given input. Each constructor accepts the WKT, the WKB, the GML, and the ESRI shape [ESR98] representation either as a CLOB for the textual formats or as a BLOB for the binary formats. The differentiation between the various formats is done based on the encoded data itself.

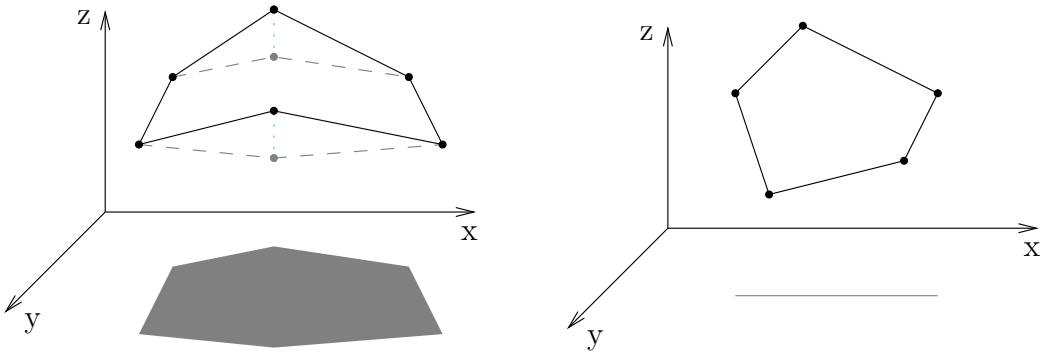
Two constructor functions are implemented in C++, one for the WKT and one for the WKB format. Each function deals with all geometry types. The format is parsed and the geometry properties are validated. The following properties are verified in that processes for polyhedra:

- The linestrings that form the rings of a facet are closed and do not intersect each other and within itself.
- A polygon is planar.
- The single polygons of a multi-polygon do not intersect, unless the intersection consists of points or linestrings only.
- The shells of polyhedron or multi-polyhedron are closed and do not intersect each other.

Once the input is deemed to represent a proper geometry, additional attributes like the area of polygons, the length of linestrings, and the values for the minimum bounding box are extracted and returned to the database engine together with the internal representation of the geometry. The transform function *GeomFromUdf3D* creates a new

ST_Geometry value by instantiating the respective subtype. The attributes of the geometry are subsequently populated with the values returned by the external code. A consistent geometry value is produced as result.

Relying on a new C++ function allows the evaluation of geometric properties beyond the scope of the property verifications already implemented in DB2 Spatial Extender for 2D geometries. Due to the fact that the Extender only operates in $\mathbb{R}^2$, it does not require that a polygon is planar in $\mathbb{R}^3$. Instead, only the projection of the polygon onto the X/Y-plane is referred to for all operations, which is planar per definition. As a consequence, the Extender does not allow the creation of a polygon that is orthogonal to the X/Y-plane because its projection would only be a linestring. *Figure 5.12* demonstrates these two critical problems for 3D operations. The grey geometries show the projection of the polygons onto the X/Y-plane.



(a) Non-planar (b) Orthogonal to X/Y-plane

Figure 5.12: Differences of polygons in 2D and 3D space

We solve the issues shown in Figure 5.12 by a 3D-specific implementation of the factory function, named *GseGeometry3D*. That way, polyhedra and multi-polyhedra can be constructed in addition to all other geometries.

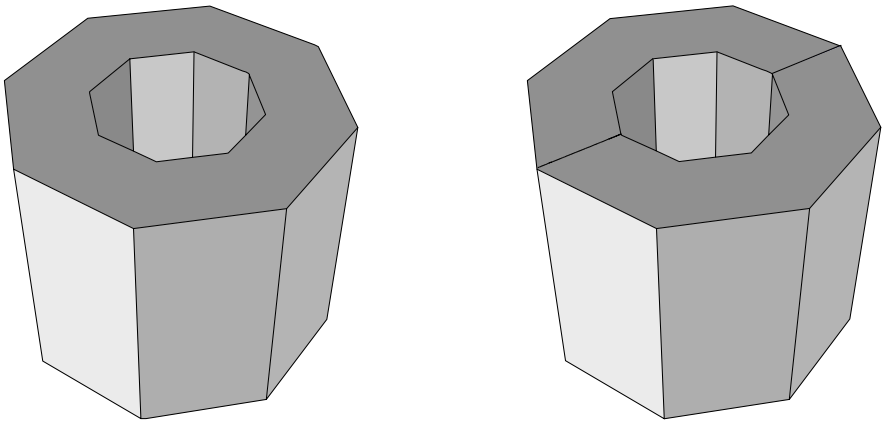
We achieve a seamless integration of the 3D-specific constructor functions just by replacing the (overloaded) *ST_Geometry* functions using the mechanism shown in Figure 5.11. The original function is renamed to *GseGeometry2D*, and a wrapper named *ST_Geometry* decides based on the numeric identifier of the SRS whether *GseGeometry3D* or *GseGeometry2D* is to be used. Additionally, new constructor functions *ST_Polyhedron* and *ST_MultiPolyhedron* are implemented. The constructors for all other data types can remain unchanged as they are based on *ST_Geometry* and merely adjust the static type of the result. *Listing 5.13* shows the SQL code for the changes to the factory function *ST_Geometry* and the definition of *ST_Polyhedron* for the well-known text representation. It is assumed that the original factory function was already renamed to *GseGeometry2D*.

```
CREATE FUNCTION db2gse.GseGeomFromText3D (
      wkt CLOB(2G), srsId INTEGER,
      xOffset DOUBLE, yOffset DOUBLE, xyScale DOUBLE,
      zOffset DOUBLE, zScale DOUBLE,
      mOffset DOUBLE, mScale DOUBLE )
   RETURNS db2gse.ST_Geometry
   EXTERNAL NAME '3d_Extender!getGeometryFromText3D'
   LANGUAGE C  PARAMETER STYLE SQL
   DETERMINISTIC  THREADSAFE  NOT FENCED
   NO SQL  NO EXTERNAL ACTION
   TRANSFORM GROUP TransformGroup3D;

CREATE FUNCTION db2gse.ST_Geometry (
      wkt CLOB(2G), srsId INTEGER )
   RETURNS db2gse.ST_Geometry
   LANGUAGE SQL  READS SQL DATA
   DETERMINISTIC  NO EXTERNAL ACTION
   RETURN CASE
      WHEN srsId / 1000000000 = 1
      THEN ( SELECT db2gse.GseGeomFromText3D(wkt, srsId,
                       s.x_offset, s.y_offset, s.x_scale,
                       s.z_offset, s.z_scale,
                       s.m_offset, s.m_scale)
               FROM   db2gse.st_spatial_reference_systems s
               WHERE  srsId / 1000000000 = 1 AND
                      srsId = s.srs_id )
      ELSE ( SELECT db2gse.GseGeometry2D(wkt, srsId)
               FROM   sysibm.sysdummy1
               WHERE  srsId / 1000000000 <> 1 )
   END;

CREATE FUNCTION db2gse.ST_Polyhedron (
      wkt CLOB(2G), srsId INTEGER )
   RETURNS db2gse.ST_Polyhedron
   LANGUAGE SQL  READS SQL DATA
   DETERMINISTIC  NO EXTERNAL ACTION
   RETURN TREAT ( db2gse.ST_Geometry(wkt, srsId)
            AS db2gse.ST_Polyhedron );
```

Listing 5.13: Defining the constructor functions

We explained in Section 5.3 that a polyhedron consists of a set of facets, grouped in shells, and each facet is a polygon that can have multiple rings. All rings, except the

first ring represent holes in the polygon. The 3D Extender does not support facets with holes to reduce the complexity of the implementation. This restriction does not impose a loss in functionality because each polygon with holes can be divided into two (or more) polygons with no holes as *Figure 5.13* shows. Thus, the facets in a polyhedron shell must always be defined by a single ring.



(a) Facet with hole (b) Facet without hole

Figure 5.13: Resolving holes in facets of polyhedron shells

Another aspect derived from CGAL is that points defining a facet have to be specified in a counter-clockwise order as seen from the outside of the polyhedron. This requirement is more restrictive than DB2 Spatial Extender used to be, which allows both clockwise and counter-clockwise ordering. With the enforced ordering, CGAL can easily determine the interior of he polyhedron (and also the interior of the facet). *Figure 5.14* illustrates the order in which the points have to be given. The circular arrows indicate the counter-clockwise ordering. For example, the facet at the front of the cuboid is correctly defined by P_1, P_2, P_3, P_4.

Using a clockwise direction prevents the construction of the polyhedron in CGAL, and the constructor function is bound to return an error message to the caller. Given that polyhedra are usually constructed by applications and not by manually typing in the WKT or WKB representation, this restriction does not lead to considerable problems, especially not for the 3D Extender.

Implementing other Functions

The two methods *ST_AsText* and *ST_AsBinary* convert a geometry to the well-known text or well-known binary representation, respectively. Both methods are inherited from the root type of the spatial hierarchy and overriding them for the ST_Polyhedron and ST_MultiPolyhedron types appears to be the simplest approach to handle the conversion

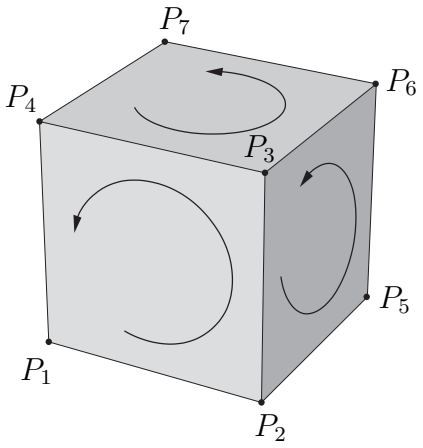


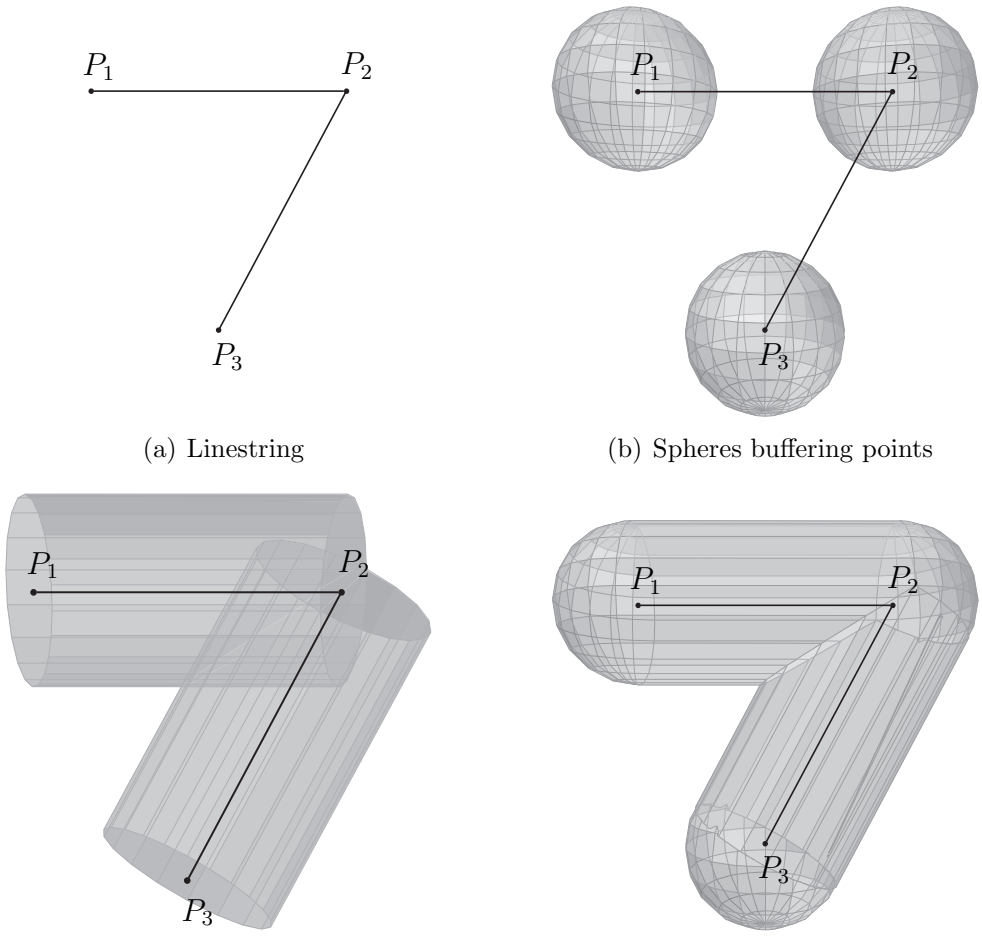
Figure 5.14: Order of points on a polyhedron shell

of the new data types to the formats described in Sections 5.3.1 and 5.3.2. However, we also adapted the storage model and binary encoding of all other data types if the geometry is in $\mathbb{R}^3$. Thus, overriding does not completely solve the problem, unless the methods are overridden for all subtypes of ST_Geometry. That defeats the purpose of method overriding. Instead, we redefine the methods at the root of the spatial type hierarchy to call the 3D Extender to handle the CGAL-encoding for geometries in $\mathbb{R}^3$ and the DB2 Spatial Extender for geometries encoded in the original internal representation. A `CASE` expression is injected as Figure 5.11 proposed.

An advantage in Bittner [Bit05] is that the methods *ST_AsText* and *ST_AsBinary* did not have to be reimplementation for the non-3D data types as described. The storage model of all non-polyhedron geometries was not modified. The differentiation between 3D and non-3D functionality was accomplished by the data types only. Therefore, the original implementation of the conversion method was still applicable to the non-3D types and the method *ST_AsText* could be defined as an overriding method for ST_Polyhedron and ST_MultiPolyhedron only. The price for a simpler implementation of *ST_AsText* has to be paid at runtime if calculations with points, lines, and polygons are to be done in $\mathbb{R}^3$. Then two additional conversion steps become necessary as the DB2 Spatial Extender's internal representation needs to be transformed into a known format like WKB first and the WKB is then sent to the 3D-specific C++ function. Inside the function, the second conversion is required to build the CGAL data structures. As we show in Section 5.5.5, the conversion can be the dominating performance factor for spatial operations in $\mathbb{R}^3$.

ST_Buffer is a method that generates a new geometry from the input geometry and a given radius. The new geometry represents all points of the data space whose distance to any point of the original geometry is smaller than or equal to the radius. We redefine the method for the type ST_Geometry for the same reasons as *ST_AsText*, i. e. non-polyhedra in $\mathbb{R}^3$ use a different storage model that must be taken care of.

CGAL does not offer functions to create a buffer around a Nef-polyhedron. Thus, the logic had to be implemented by the 3D Extender itself. Three C++ functions were added for that purpose. The first function generates a sphere buffering a point and the second creates a cylinder around a linestring. The third function creates a polyhedron that encloses a polygon by parallel shifting the polygon perpendicular to the polygon plane in both directions by the given distance (radius). The results of the last two functions are not yet the complete buffers for linestrings and polygons because the linestrings and points of the polygon boundary are not properly handled yet. The logic takes the boundary, which is a linestring, and creates a buffer for it and joins this buffer with the previous polyhedron. Likewise, the buffer for a linestring consists of the cylinders that are merged with spheres as buffers around the points of the linestring. The merging is a complex operation and CGAL's union algorithm is used for that. *Figure 5.15* depicts this process for a linestring.



(a) Linestring

(b) Spheres buffering points

(c) Cylinders buffering line segments

(d) Union of all buffers

Figure 5.15: Creating buffer around a linestring

The approach to create a buffer for any geometry in $\mathbb{R}^3$ is by no means optimal. Consequentially, it leads to long execution times. Some performance numbers of the 3D Extender are presented in Section 5.5.5.

Computing the intersection of two geometries in $\mathbb{R}^3$ is directly supported by CGAL as long as the geometries are available as Nef-polyhedra. That way it is possible to intersect points, linestrings, polygons, and polyhedra pairwise with each other, regardless of the specific geometric type. The method *ST_Intersection* is replaced by a wrapper function that calls CGAL for geometries in $\mathbb{R}^3$ and the original DB2 Spatial Extender implementation otherwise. The result of an intersection in $\mathbb{R}^3$ can be any combination of a set of all geometric primitives. Although DB2 Spatial Extender only supports homogenous collections, the result produced by the 3D Extender can be heterogenous and they are stored as such in the ST_GeomCollection type. Thus, the enhancement broadens the functionality of the DB2 Spatial Extender in this respect.

The function *ST_Intersects* has a dedicated implementation in DB2 Spatial Extender. That is not the case in CGAL, so the logic of the function is based on the intersection operation. The function returns 1 (one) if the intersection is not empty, and 0 (zero) otherwise. The intersects operation is the most important one to implement spatial joins in a relational database system, i. e. the capability to use spatial comparison operations as join criteria. It should be noted that the functions *ST_Crosses* and *ST_Overlaps* are virtually identical to *ST_Intersects* (except for restrictions on the input geometries). Thus, both functions are available as well.

The last two methods defined in the 3D Extender calculate the area covered by the facets in the shells of a polyhedron and the volume of a polyhedron. The method *ST_BoundingArea* is defined on the types ST_Solid and ST_MultiSolid. Its specific implementation for the polyhedron and multi-polyhedron types is achieved by overriding it for ST_Polyhedron and ST_MultiPolyhedron, respectively. Once the polyhedron is represented in CGAL, its shells can be extracted and the area calculated for each facet by triangulating it and adding up the areas of the triangles. The method *ST_Volume* is handled in a similar fashion. It tetrahedronizes the polyhedron, calculates the volume for each tetrahedron and sums up all the results.

5.5.4 Spatial Indexing

The SQL/MM spatial standard does not define any mechanisms with respect to indexing spatial data. In that it follows the paradigm of the SQL standard [ISO03i], which considers indexing as being solely in the domain of the specific database management systems (DBMSs). Therefore, our main focus for the 3D Extender was not spatial indexing either. However, managing huge amounts of data practically requires adequate indexing techniques and the 3D Extender as enhancement of the DB2 Spatial Extender provides indexing mechanisms for data in $\mathbb{R}^2$ as well as $\mathbb{R}^3$.

Providing support for a spatial index for three-dimensional data can be achieved by extending the grid index [IBM04d], as it is implemented in the DB2 Spatial Extender, with another dimension. Based on the detailed analysis of the grid index given by Walther [Wal05] an index extension that handles the X, Y, and Z coordinates is implemented.

Another approach to index three-dimensional data using the DB2 index extensions [SS04] can be built on top of interval trees [BES+05, KPS00] and space-filling curves [Sag94]. An implementation and evaluation of an index extension that computes such intervals for the minimum bounding rectangle in $\mathbb{R}^2$ on a space filling curve and stores those intervals in an interval tree can be found in [Ott06]. It showed that such an index mechanism could offer slightly better performance than DB2's two-dimensional grid index. Similar results apply for indexing of geometries in $\mathbb{R}^3$.

Yet another approach can be based on the technique described by Freytag [FFS00] where the geometries are approximated by space-filling curves like the z-ordering in such a manner that the curve can handle multiple granularities. The ordering of the curve is adjusted so that the granularities are interleaved specifically. The multiple granularity levels lead to a reduction of the index size while preventing multiple index scans to query each level. The ordering is chosen so that a consecutive range query will cover the whole space of interest. The mechanism has to be adjusted to the 3D space, however. Unfortunately, such techniques require the access to the exact geometry shape and this is not possible with DB2 index extensions.

5.5.5 Performance Results

We can only consider the 3D Extender for real-world applications if it can offer an adequate performance. The major factor will be the execution time of the 3D-specific functions. The implemented constructors and comparison functions are mainly defined through spatial set operations in CGAL, so it boils down to the performance of CGAL. In the current section we analyze the execution times of the new or modified spatial routines for growing complexities of the processed geometries. Additionally, several mechanisms for the internal storage format of the geometries are tested to determine the influence of the transformation between the CGAL objects and the serialization of the geometry when it is stored in the database system as ST_Geometry value.

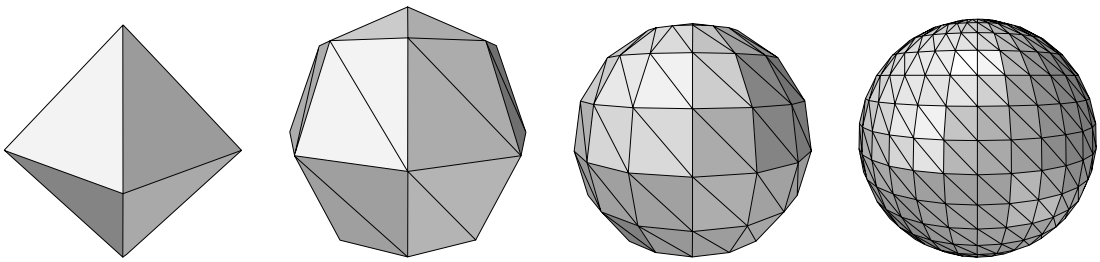
The routines to convert geometries between the database-internal structure and an external representation like WKT may have a substantial impact on the performance of an application if the database system is mostly used to store the geometries only and not to process them any further. Therefore, we measure the performance of the constructors and conversion methods for the external formats. If an application processes geometries in $\mathbb{R}^3$, it will usually select those geometries based on their spatial extent. That makes comparison operations for two geometries a necessity, especially if spatial joins on three-dimensional data are needed. The last group of functions analyzed are

the set operations that generate new geometries, for example by computing the spatial intersection or difference. For all functions, we measure the execution time depending on the complexity of the input geometries. Additionally, the spatial set operations and the constructor functions produce new geometries and the complexity and space utilization of those geometries is gathered.

All measurements use approximated spheres as the same general geometric shape. The complexity of a geometry is based on the number of facets of the sphere. This number can be influenced by a dedicated constructor function *ST_Sphere*, which takes the center point, the radius, and a detail level as input parameters. The detail level d is the means to control how many facets the resulting sphere shall have. The algorithm to construct such a sphere produces $\frac{d}{2} \cdot d$ rectangles by cutting the sphere into $\frac{d}{2}$ horizontal slices and approximating the surface of each slice with d rectangles. Each rectangle is divided into two triangles. Special considerations are necessary for the poles where two points of the rectangle collapse to a single point and the result is already a single triangle. Thus, the exact number of triangles $t(d)$ of a sphere at detail level d is:

$$t(d) = 2\left(\frac{d}{2} \cdot d\right) - 2d = d(d-2) \tag{5.1}$$

The resulting spheres have a quadratic complexity of $O(d^2)$ depending on the detail level d. The detail level must be an even number and larger than or equal to 4. Otherwise, the above formula would not result in a positive integer for the number of generated triangles. *Figure 5.16* shows some examples for approximated spheres with different detail levels. The special treatment of the poles of the sphere is especially visible in Figure 5.16(b).



(a) $d = 4$ (b) $d = 8$ (c) $d = 16$ (d) $d = 32$

Figure 5.16: Approximated spheres with different detail levels

We employed a ThinkPad T30 with a 2 GHz Intel processor and 1 GB of physical main memory for the performance measurements. Linux with the kernel release 2.6.13 was used as operating system, and DB2 UDB Version 8.2.2 (i. e. FixPak 9) provided the relational database system platform for the tests of the 3D Extender.

Theoretical Complexity

Before the CGAL library can actually operate on the geometries, it is necessary to construct the respective CGAL object, i. e. the Nef-polyhedron. Bittner [Bit05] used the WKB representation as storage model in the RDBMS, which required a conversion from WKB to a Nef-polyhedron via a regular CGAL polyhedron. The point coordinates are stored redundantly in the WKB format where each point occurs in at least three facets. The CGAL-internal representation for polyhedra and Nef-polyhedra is an index-based data structure where each point is defined once and then only references to that point are made. Therefore, the conversion from WKB to Nef-polyhedra needs to create an array of all distinct points in the geometry. A balanced red-black tree [CLR01] is employed for the construction of the point array. The subsequent reconstruction of the facets based on the point array takes $O(\log n)$ time for each point. Thus, the cost for the construction of a polyhedron from the WKB is in $O(n \log n)$.

The last step in the process to construct a Nef-polyhedron is the conversion of the polyhedron to such a Nef-polyhedron. Nef-polyhedra are based on a complex data structure that must be derived from the index-based representation. CGAL implements kD-Trees [Ben75] to support, for example, fast point location queries. With such trees, the worst case complexity for the construction of the Nef-polyhedron amounts to $O(n^2 \log n)$. This last conversion dominates the conversion from WKB to the polyhedron and it contributes significantly to the time complexity of all operations involving Nef-polyhedra.

A different storage model was implemented in the 3D Extender by directly serializing the CGAL-internal format and then compressing it. We added the compression itself because the CGAL-internal representation contains a lot more explicit information about the relationships between points, edges, facets and shells and, thus, is much more verbose. That verbosity leads to a higher space utilization and compression algorithms like *deflate* [DG96] or *bzip2* [Man99, Sew05] reduce the space requirements in exchange for computation time to about $\frac{1}{3}$. Summarized, the theoretical costs for the construction of the Nef-polyhedron are only linear $O(n)$ for the decompression and the actual construction. That is quite a significant reduction of the complexity compared to $O(n^2 \log n)$ with the WKB storage.

CGAL's comparison functions in $\mathbb{R}^3$ are based on the spatial set operations for Nef-polyhedra. Hachenbacher [HK05] has shown that the computation of the intersection of two Nef-polyhedra can be performed in $O(n \log^3 n)$ time and the described logic is also implemented in CGAL. If a 3D Extender routine is to return a geometry, for example *ST_Intersection*, then the implementation in [Bit05] resulted in another step for the backward conversion of the Nef-polyhedron to the corresponding WKB representation that required another $O(n^2 \log n)$ time. Thus, it is easily possible that the majority of the time is spent solely with the conversion of the input geometries and the result. The 3D Extender performs the serialization and compression in $O(n)$, avoiding this substantial overhead.

Conversions from and to External Formats

The population of three-dimensional objects, in particular polyhedra and multi-polyhedra, in a spatial database is achieved by invoking the constructor or factory routines and inserting the generated geometries into the intended target table. The performance of the constructors is not only related to the internal storage model of the 3D Extender but also to the complexity of the geometries being handled. Our comparisons included four different storage models:

1. using the well-known binary format as proposed by [Bit05],
2. serializing the CGAL Nef-polyhedra to an uncompressed text stream,
3. serializing the CGAL Nef-polyhedra to a text stream that is compressed using the *deflate* compression algorithm, and
4. serializing the CGAL Nef-polyhedra to a text stream that is compressed using the *bzip2* compression algorithm.

We used the two different compression algorithms because *bzip2* tends to have a better compression ratio (compared to *deflate*) but it usually takes longer to compress the data. However, the subsequent measurements show that the difference is only marginal in the context of the 3D Extender because of the remaining logic that still needs to be processed, i. e. the actual spatial functionality. Storing the uncompressed Nef-polyhedra establishes a base line for the compressed storage models, but it is not an acceptable option for practical purposes because the format is just too verbose and requires too much disk space as *Figure 5.17* proves. With the fast growth of the uncompressed data stream, it already reaches the 1 MB size limit imposed by the DB2 Spatial Extender's ST_Geometry type when a polyhedron is comprised of 1000 facets. Therefore, all subsequent measurement include the uncompressed storage model only up to this size as a reference and focus on the other models.

The well-known binary representation does not include as much redundant information as the CGAL-internal format. Thus, it is more compact and the assumption that it is quite a bit slower to construct a CGAL Nef-polyhedron from it (due to the complexity of $O(n^2 \log n)$) holds true. *Figure 5.18* illustrates the construction of CGAL objects from their counterpart that is stored in a table in the relational database, i. e. a ST_Polyhedron value. The execution times based on the WKB format are indeed slower, namely 40% compared to the *deflate*-compressed storage. However, the difference is not as high as may have been expected. The reason is that the CGAL-internal serialization mechanisms is comparably slow so that subsecond execution times cannot be achieved for more complex geometries that have several thousand facets. It should be noted that the decompression does not contribute any measurable overhead since the uncompressed format comes with about the same performance as the compression with *deflate*.

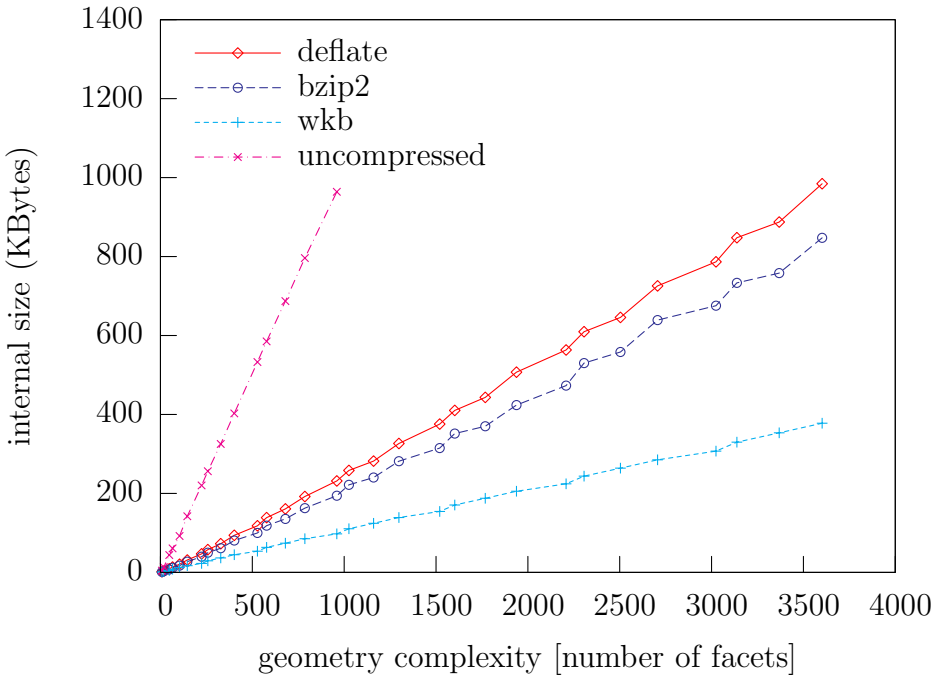


Figure 5.17: Storage requirements depending on geometry size

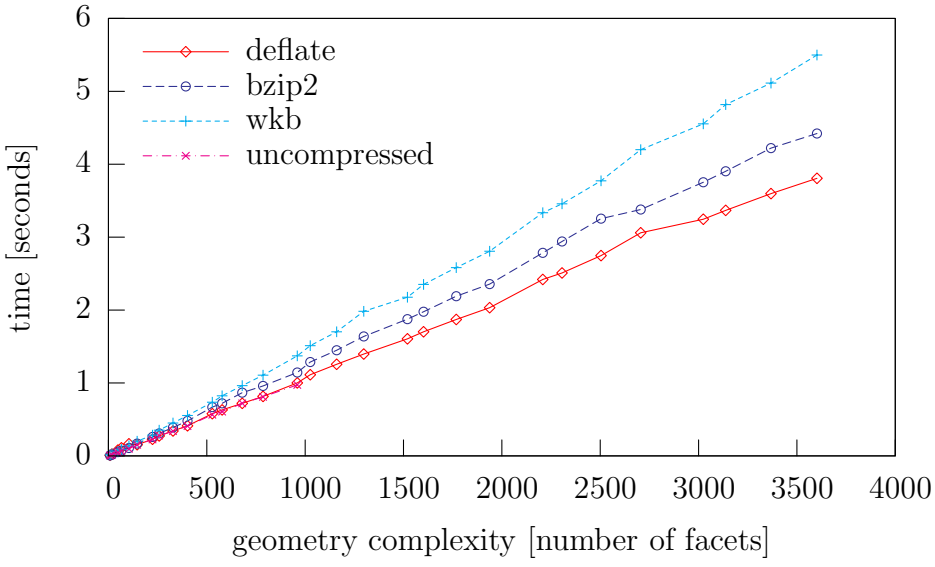


Figure 5.18: Construction of CGAL Nef-polyhedra from ST_Polyhedron values

The previous results were targeted at internal processing only. What truly matters for a user of the 3D Extender, however, is the execution time for the constructor functions like ST_Polyhedron and ST_Geometry for a given well-known text or well-known binary representation (besides the execution time for other spatial routines). Additionally, when geometries are extracted from the database, the time consumed for the conversion of the selected geometries to their WKT or WKB format is important. *Figure 5.19* shows how both routines behave with growing complexity of the geometries being processed, depending on WKT or WKB input or output.

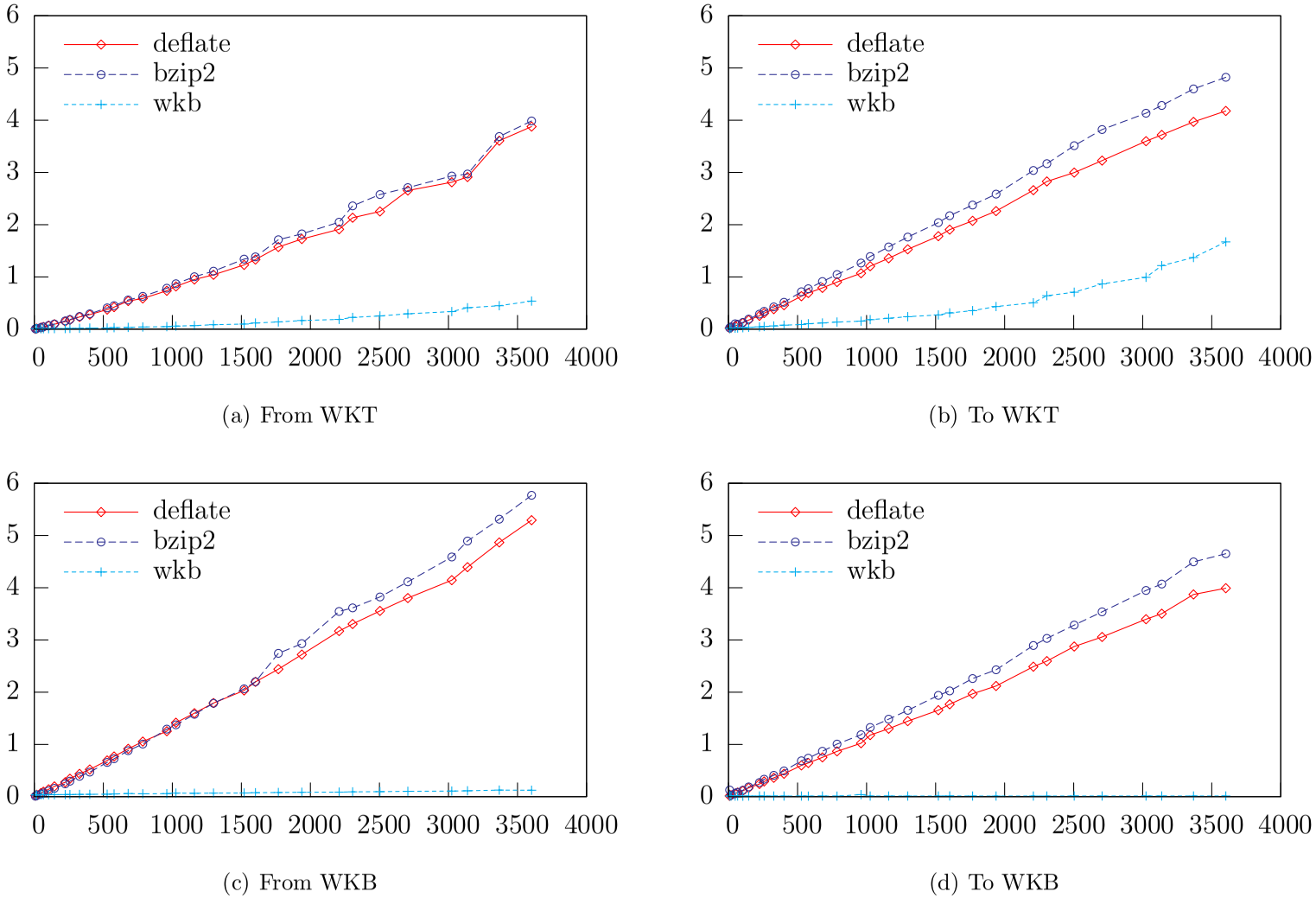
The pattern in the figures is clear. The execution times for the compressed and uncompressed CGAL-internal format are roughly equal, with the *bzip2* decompression being slightly slower than *deflate*. It matches the initial expectations. The execution times based on the well-known binary storage model way outperform all other techniques. That is due to the fact that a given WKB representation is simply stored exactly as provided. The given WKT string is parsed and only analyzed insofar as is necessary to build the well-known binary format. In particular, the CGAL Nef-polyhedron is not built and no validation of the integrity of the geometry is done, placing the other implementations at a distinct disadvantage in this respect.

Comparing Two Geometries

Once the geometries are populated in the database we can use them for regular spatial processing. That includes the test for intersection as it is the means to express a spatial join in a spatial database. The intersection tests are mandatory for two-dimensional spatial data as in a GIS but also for three-dimensional data, which is used in CAD applications, for example. A typical query in a CAD application is to determine whether any two parts of an object to be manufactured intersect, i. e. if they occupy the same space and, therefore, the object could not be assembled properly. For example, when designing a car, it must be ensured that the brake calipers do not interfere with the wheel axle and the rim.

There are four constellations for the relationships of two geometries in $\mathbb{R}^3$ that are depicted in *Figure 5.20* by using spheres. Both geometries can be identical in terms of the occupied data space (*equals*), one geometry can be fully inside the other (*contains*), both geometries can partially *overlap*, and both geometries can be *disjoint*. Equality and containment are actually a special case of the overlap. They are treated as as three distinct cases for the measurements because the execution times vary significantly depending on the amount of overlap. The shaded areas in the figures indicate the intersecting regions.

The routines that we defined for the 3D Extender take two ST_Geometry values as input, build the respective Nef-polyhedron for each and then invoke CGAL to test whether the two geometries have the requested spatial relationship or not. *Figure 5.21* shows



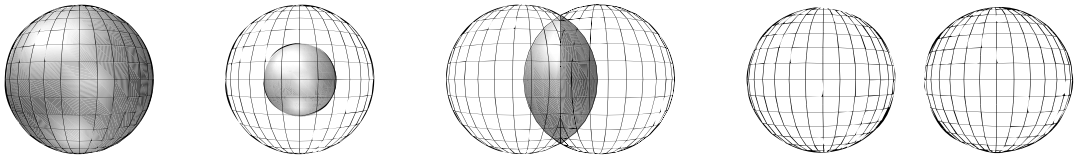
(a) From WKT

(b) To WKT

(c) From WKB

(d) To WKB

Figure 5.19: Execution time for conversion from and to external data formats



(a) equals (b) contains (c) partial overlap (d) disjoint

Figure 5.20: Tested relationships of geometries in $\mathbb{R}^3$

the performance numbers for the method *ST_Intersects*. We evaluate all four different spatial relationships between the two geometries with growing complexity of the input. The internal geometry information was compressed via the *deflate* algorithm as it offers generally the best space/time trade-off.

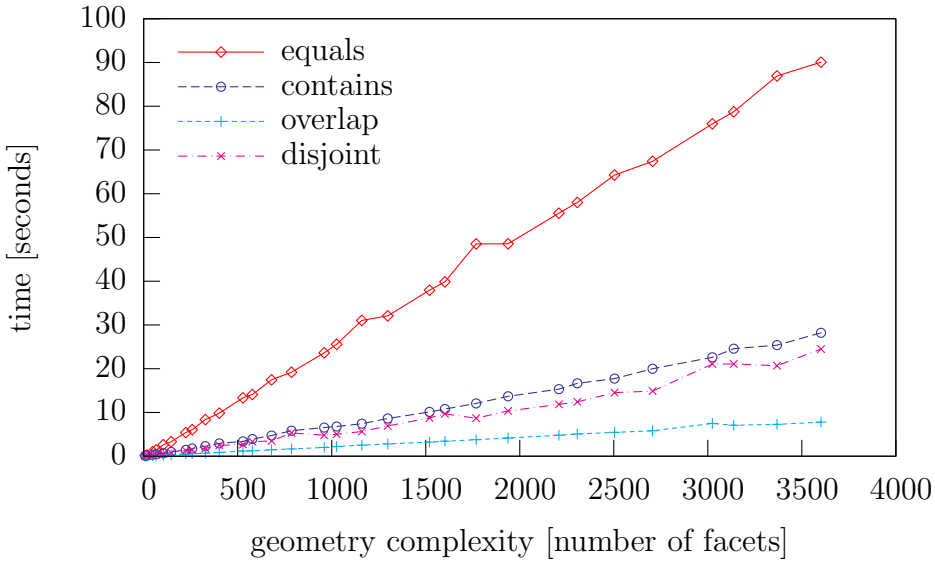


Figure 5.21: Execution time of *ST_Intersects*

The overall execution time for a single comparison is rather high. In particular, the case where both geometries are identical takes a long time with more than 1.5 minutes for the most complex geometries tested. The root of the problem can be found in the sub-optimal implementation in CGAL. The intersects function for Nef-polyhedra first computes the complete intersection and then determines whether the result is empty or not. An empty result only occurs for disjoint geometries.

An optimization based on the minimum bounding boxes (MBB) of geometries can be implemented. Testing the boxes for overlap can quickly indicate if there is the potential for overlap in the first place. If the boxes do not intersect, then the respective geometries will not intersect either. The MBB test can be implemented in the external code that

communicates directly with CGAL or it could be based on the geometry attributes that are already extracted when the geometry is constructed. *Figure 5.22* compares both with the native implementation that does not take advantage of the MBB information. Determining the MBB in the external code requires that the CGAL structures are built to extract the minimum and maximum X, Y, and Z coordinates of the geometry. That step takes the majority of the time. However, the CGAL structures need to be built anyway if the MBBs do overlap. The only relevant overhead in this case arises from the extraction of the coordinates. Testing the MBBs directly on the database side using the attributes stored in the ST_Geometry data type before even the external code is used is the best option for disjoint geometries. The external routine does not need to be invoked at all and no time is wasted to construct the CGAL structures.

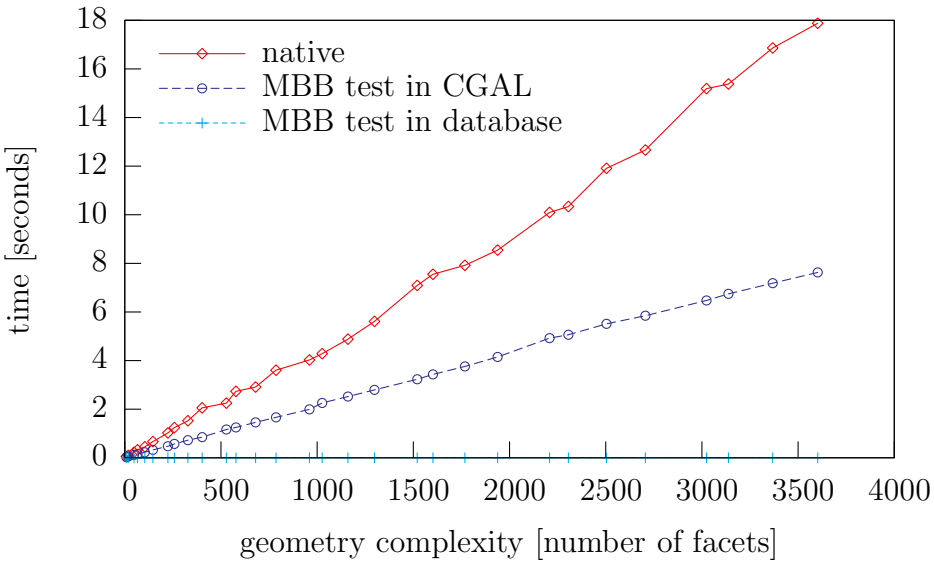


Figure 5.22: Optimizing *ST_Intersects* by testing MBBs

Another improvement can be based on an early-out mechanism. As soon as the first intersection is found, the computation can be terminated and the result is known, regardless of the remaining shape of the not-yet computed intersection. A simple implementation to simulate this optimization iterates over the points of one geometry and tests if one of those points lies in the interior of the other geometry. We compare this algorithm in *Figure 5.23* against the native implementation in CGAL where both input geometries partially overlap. The bottom line is that the determination whether both geometries intersect is significantly faster and the dominating factor becomes the construction of the CGAL objects alone.

The improvement using point containment does not speed up the processing if two disjoint geometries are being compared or if both geometries are defined in such a way that no point of one geometry lies inside the other. In such cases, the logic only adds over-

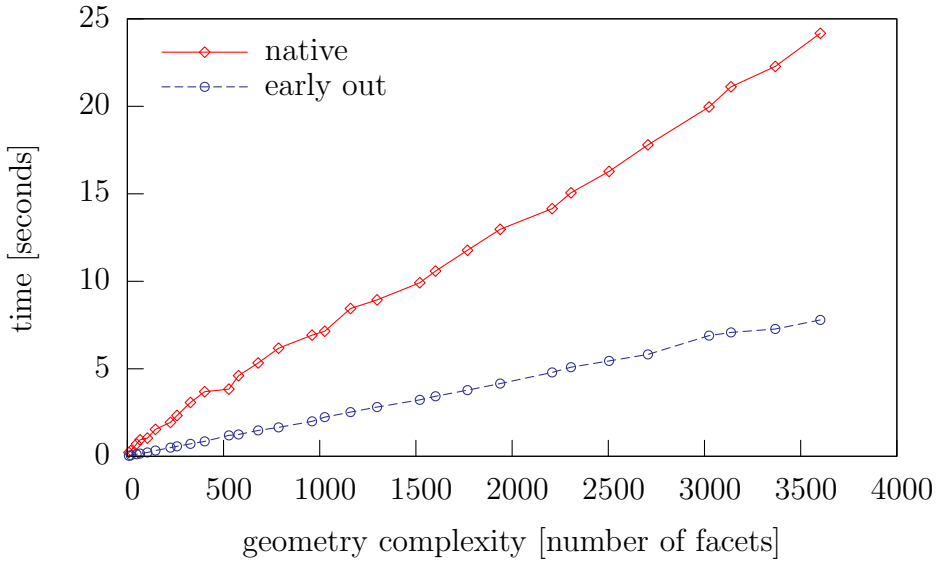


Figure 5.23: Optimizing *ST_Intersects* by testing for point containment

head and the actual intersecting region is still constructed to come up with a definitive answer. Thus, a combination of the MBB test and the point containment should be adopted to cover more situations.

The comparison of the MBB on the SQL side can be considered as mandatory and it is a known technique for the two-dimensional data already managed today by the DB2 Spatial Extender. This technique is the foundation for the spatial grid index [IBM04d, Wal05]. The point containment test can be added to the CGAL-internal processing logic by implementing a dedicated function that tests for an intersection. Additional optimization may be included there as well as the point containment is not necessarily optimal in all cases.

So far only the *ST_Intersects* function was evaluated with respect to its performance and possible improvements for it. Other comparison routines like *ST_Contains* and *ST_Equals* show the same or very close execution times and the results and optimizations are directly applicable. Especially the MBB test can be exploited for the equality test as two geometries cannot be equal if their MBB is not equal.

Generating new Geometries

Another major group of routines defined by the SQL/MM spatial standard and the DB2 Spatial Extender is comprised by the routines that generate new geometries from others. That includes not only methods like *ST_Boundary* and *ST_Buffer* but also spatial set operations. The methods that return a geometric property are often not much more complicated than quickly gathering some information from the input geometry and then

building the resulting geometry as in the constructor functions. For example, the method *ST_Boundary* applied to surfaces requires a change of the data type to a curve and all facets need to be removed from the CGAL-internal representation. Direct access is available in CGAL to both information. Determining the centroid with the method *ST_Centroid* is based on the minimum bounding box (MBB). A traversal of all points in the geometry gives the coordinates for the MBB and then the centroid point is directly calculated and constructed. Thus, such operations are only bound by the construction of the CGAL Nef-polyhedra.

Spatial set operations are the basis for the comparison functions available in CGAL. The execution times for methods like *ST_Union* are very similar to the group described before. The major difference can be found in the returned data where no integer value is produced but rather another geometry. In such cases, the resulting CGAL structure needs to be converted to the internal representation used by the 3D Extender, i. e. serialized and compressed. *Figure 5.24* compares the execution times of the function *ST_Intersection* for the four types of relationships of the two input geometries. Comparing those results with Figure 5.21 reveals the additional time spent on the resulting geometry. The figure also shows that the computation of the intersection for two identical geometries is very complex and time consuming. That is intuitively understood as the maximum possible number of intersecting facets exists in that situation.

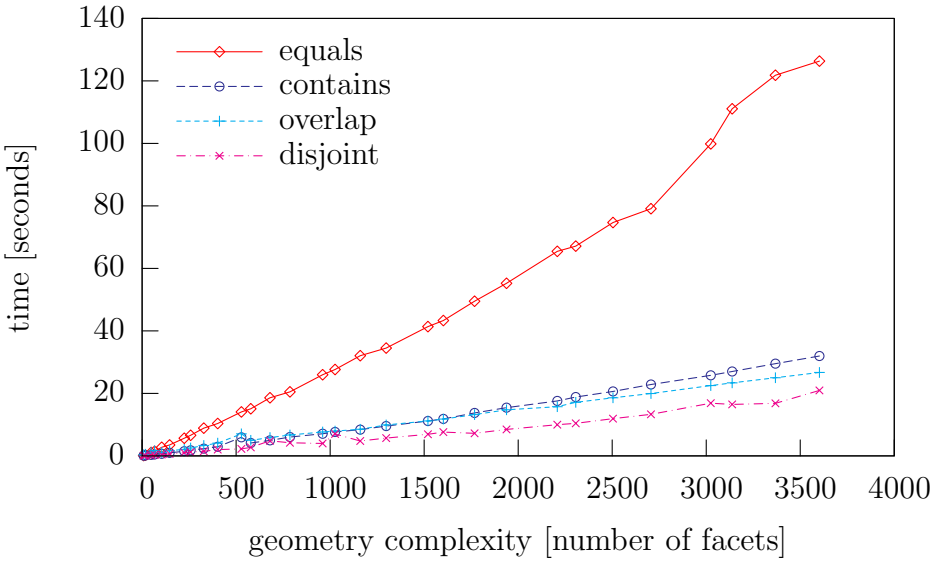


Figure 5.24: Execution time of *ST_Intersection*

Some of the optimization that we discussed for the comparison functions do also apply for the spatial set operations. For example, the union of two geometries with disjoint MBBs is a direct combination of both geometries without the requirement to test for any common regions at a more detailed level. CGAL does not support such a short-

cut so that the union of two disjoint geometries takes more than 55 seconds in total if both geometries have about 3600 facets. Likewise, the intersection of disjoint geometries is empty if the intersection of the MBBs is already empty. Nevertheless, CGAL tries to compute the intersection on the detailed geometries, leading to an unnecessary high execution time as *Figure 5.25* proves. The optimized versions determine based on the MBBs that no intersection exists and return an empty geometry right away. The fastest implementation determines the empty intersection using the MBB stored in the attributes of both ST_Geometry values. Using the external code for that test looses the time during the construction of the CGAL structures. The actual test of the MBBs is insignificant in comparison to the construction.

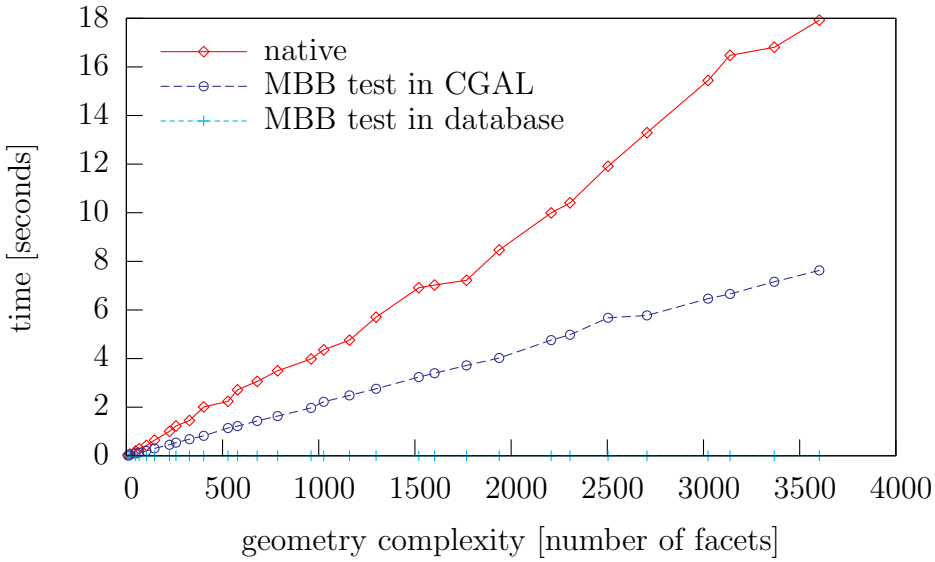


Figure 5.25: Optimizing *ST_Intersection* by testing MBBs

We explained the logic for the creation of a buffer for a geometry in $\mathbb{R}^3$ in Section 5.5.3 already. Essentially, we take the geometry apart, produce a sphere around each point, a cylinder around each edge and shift surfaces and facets perpendicular to their plane into both directions to construct a volume (polyhedron). The union of all these three-dimensional geometries forms the result, i. e. the buffer around the input geometry. The geometries to be combined are all partially overlapping with each other. Although overlapping geometries yields the best results as *Figure 5.26* demonstrates, spending 20 seconds (excluding the construction time) to union just two geometries with 3600 facets cannot be considered as acceptable execution time. Therefore, improved algorithms are a must for the union operation itself and even more so for *ST_Buffer*.

As a final remark we mention that the results of the spatial set operations measured above are not more complex than the input geometries. The resulting polyhedra have at most $n + m$ facets if the two input geometries have n and m facets, respectively.

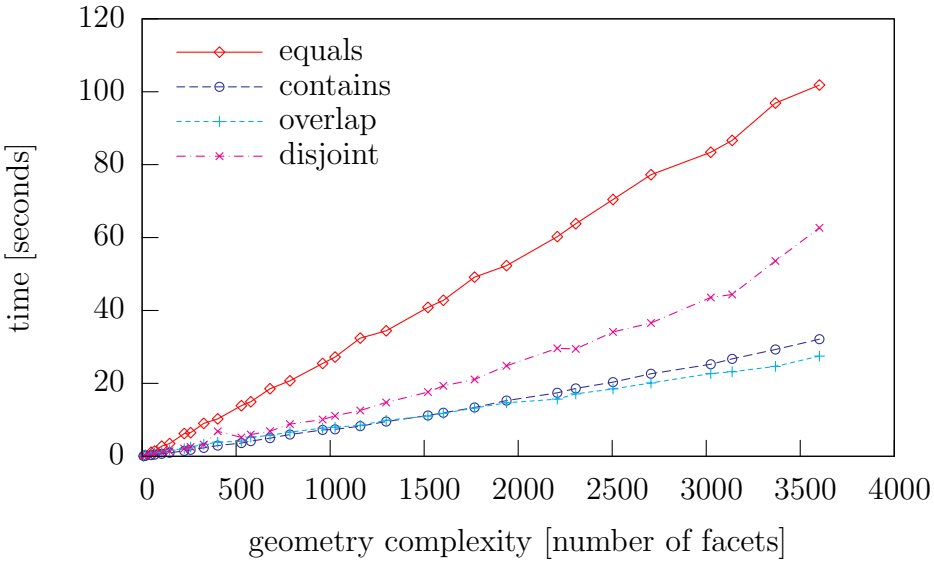


Figure 5.26: Execution time of *ST_Union*

Figure 5.27 compares the complexity of the result with the combined complexity of the input geometries. Only a selected subset of operations is shown, but the omitted ones follow exactly the same pattern. Each curve in the figure describes the operation that was performed (difference, intersection, or union) and the relationship of the two input geometries (overlap, contains, equals, or disjoint). In all cases a linear growth relative to the input is apparent.

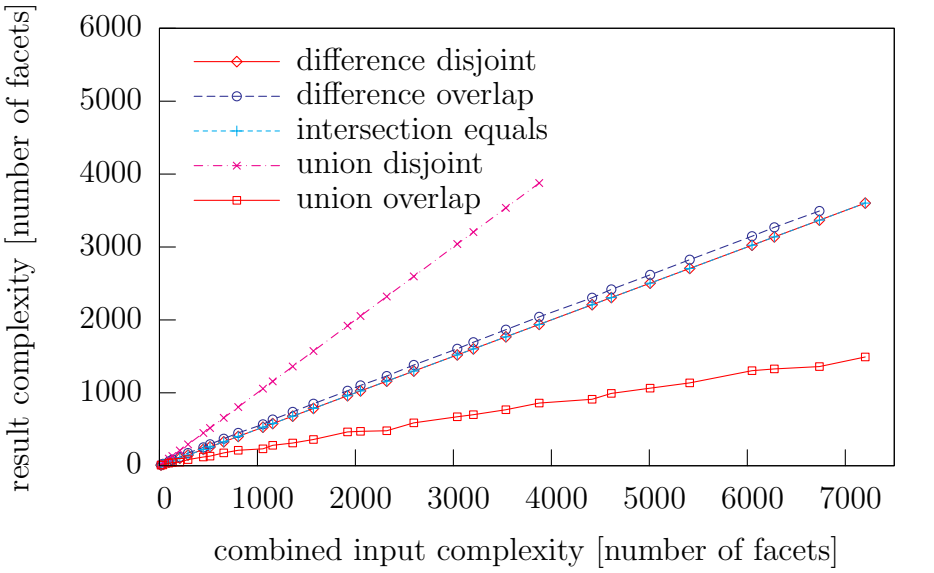


Figure 5.27: Complexity of results from spatial set operations

Other Routines

The last group of routines defined for the 3D Extender collects all methods and functions that do not easily fit into the other three groups evaluated before. Methods like *ST_IsValid*, *ST_Volume*, or *ST_NumFacets* belong here.

ST_IsValid is an important function for the WKB storage model. The constructor functions for that model do not build the CGAL representation of the geometry and do not perform any consistency checks like testing for the closure of a polyhedron. Therefore, it is easily possible to create invalid geometries from an incorrect or incomplete WKB specification. The SQL/MM spatial standard defines the method *ST_IsValid* to detect any inconsistencies and so does the 3D Extender. The other presented storage models do construct the CGAL structures, which includes the additional steps for the validation right during the construction phase. For comparison, the method *ST_IsValid* is fully implemented for all storage models, i. e. the CGAL structure is generated and the CGAL method *is_valid* invoked. The time needed for the complete process is is shown in *Figure 5.28*.

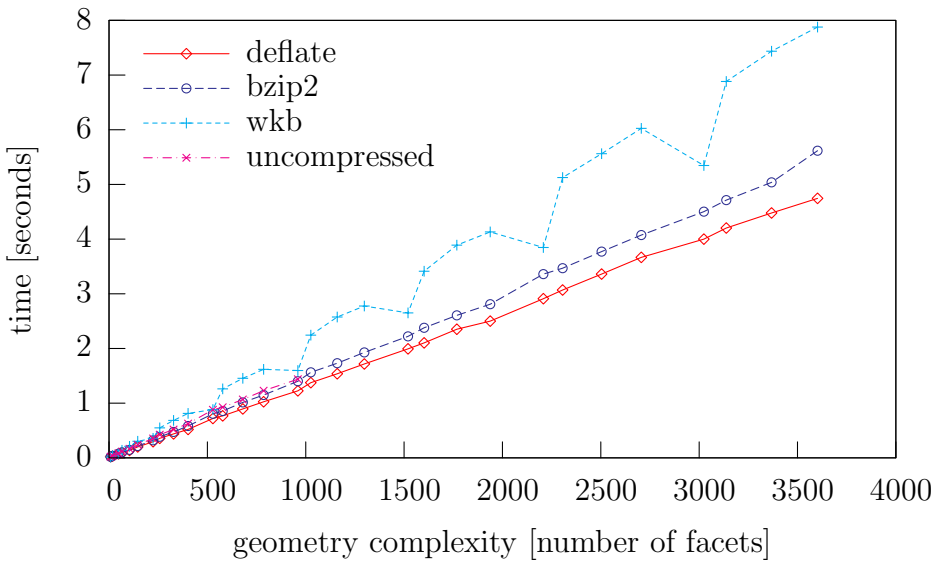


Figure 5.28: Execution time of *ST_IsValid*

The storage models based on the serialization and (optional) compression of the CGAL-internal representation out-perform the WKB approach significantly, even with the same amount of work to be done for the actual validation. We can attribute the time difference to the construction of the respective CGAL object. The regular peaks in the curve for the WKB representation are due to CGAL-internal optimizations, which require additional CPU cycles to complete. Taking advantage of the validation in the constructor function for the non-WKB storage models would actually lead to the situation that no time at

all is needed in *ST_IsValid* in an SQL statement – except the compilation phase for the SQL statement itself – as no validation is necessary and the routine could always return 1 (one) as its result.

Other methods like *ST_Volume* or *ST_NumFacets* have a similar implementation as *ST_IsValid*. CGAL provides the functionality for most methods of the 3D Extender. For example, the class for Nef-polyhedra comes with the method *number_of_facets* which directly maps to *ST_NumFacets*. However, not all routines are directly supported this way. For instance, there is no CGAL method to return the volume of a three-dimensional object, measured in the units of the underlying coordinate system. Such logic is added in the code that connects CGAL with the database system. To calculate the volume, we convert the CGAL Nef-polyhedron to a CGAL polyhedron as described in Section 5.5.1. Then the polyhedron can be tetrahedronized using CGAL's functionality. Calculating the volume for each tetrahedron is straight-forward and the single results are summed up to get the volume of the original polyhedron.

5.5.6 Visualizing Three-Dimensional Objects

The final part to complete the implementation of the 3D Extender and to verify its proper functioning is a tool to visualize the three-dimensional geometries. We implement a viewer to extract all spatial data from a table and depict the geometries. The functionality of such a viewer goes beyond the scope of the SQL/MM spatial standard, even including the proposed enhancements for spatial data in $\mathbb{R}^3$. In fact, non of the SQL-related standards ventures into the realm of graphical interfaces.

The viewer for the 3D Extender is written in Java and incorporates a library for visualization and handling of 3D objects, i. e. Java3D [Sel02]. The viewer provides a simple interface to access all tables with spatial data. The data is read from the table and converted to its WKB representation during the extraction process using the method *ST_As Binary*. Thus, the viewer is completely independent of the actual implementation of the logic for the three-dimensional processing. Once the data is available in the application layer, the required Java3D structures are built and shown in the visualization frame. Accessing any spatial table even allows the inclusion of spatial functionality like buffering and intersection by querying a view like VIS_EXAMPLE in *Listing 5.14*.[4]

The view represents only a single but complex three-dimensional geometry. The geometry was constructed by creating buffers around seven points, resulting in seven approximated spheres. Each point is given by its X, Y, and Z coordinate. The six smaller spheres (radius 5) are joined (spatial union). No overlapping region exists between those spheres, so the joined geometry is a multi-polyhedron comprised of six parts. The final step is to intersect the largest sphere (radius 10) with the multi-polyhedron, returning

[4] DB2 requires the double-dot notation for method invocations. That is not consistent with the SQL:2003 standard, which mandates only a single dot.

```
CREATE VIEW vis_example(geometry) AS
   VALUES ST_Point(50, 50, 50)..ST_Buffer(10)..
      ST_Intersection(
         ST_Point(60, 50, 50)..ST_Buffer(5)..
         ST_Union(ST_Point(50, 60, 50)..ST_Buffer(5))..
         ST_Union(ST_Point(40, 50, 50)..ST_Buffer(5))..
         ST_Union(ST_Point(50, 40, 50)..ST_Buffer(5))..
         ST_Union(ST_Point(50, 50, 60)..ST_Buffer(5))..
         ST_Union(ST_Point(50, 50, 40)..ST_Buffer(5)));
```

Listing 5.14: View to construct a sample 3D object

another multi-polyhedron. The final geometry is shown in *Figure 5.29*, which is actually extracted from a screen-shot made from the viewer. The three-dimensional object was rotated by the viewer using Java3D capabilities to provide a better view of the result.

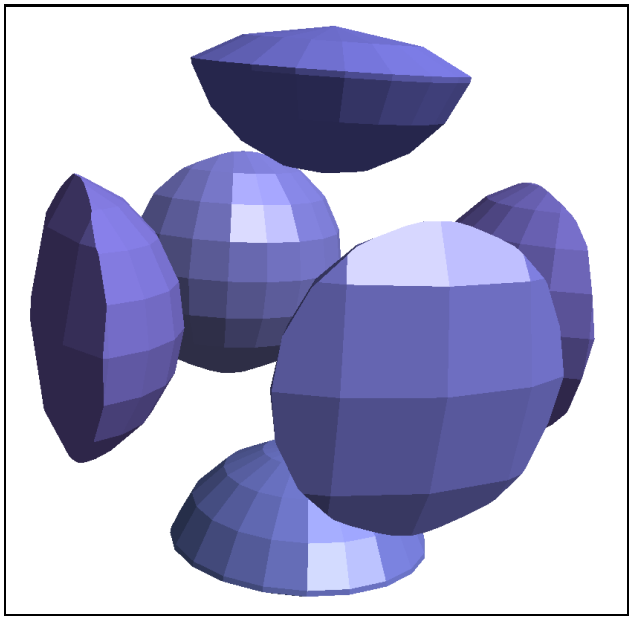


Figure 5.29: Geometry rendered by the 3D Extender visualizer

Viewing and visualization functionality should be made available in components like the DB2 Control Center. The DB2 Control Center is a tool to manage DB2 database systems, and it also offers facilities to query the data in the database. Presenting the spatial data in a text or binary format is often not as informative as a picture or map. With the higher complexity of three-dimensional objects, that becomes even more true.

5.5.7 Conclusions for the 3D Extender

Our implementation of the prototypical 3D Extender shows that the proposed extension for the SQL/MM spatial type hierarchy can easily be implemented. The new spatial data types for three-dimensional objects fit seamlessly into the existing type system.

Basing the 3D Extender on an existing, closed-source product did impose serious obstacles resulting from the internal and not documented storage model used by the DB2 Spatial Extender. The original definition of the spatial types and methods using the infrastructure provided by DB2 itself opened a way to add new logic. We could redefine existing methods and employ the evaluation of a simple condition to determine whether the original DB2 Spatial Extender logic was to be invoked or if the new 3D-specific logic should take effect.

Deciding which code branch to execute (based on the numeric identifier of the spatial reference system (SRS) that is attached with each and every geometry) did not add a measurable overhead. This technique to use different logic for the spatial processing is already well established. DB2 Spatial Extender implements a similar approach in order to provide operations in Cartesian reference systems as well as geodetic operations on the surface of a spheroid.

The adoption of CGAL as the means to perform the spatial operations in $\mathbb{R}^3$ did work out well from a functional point of view. We could implement all intended routines with relatively small effort.

From a performance and productizing point of view, CGAL is not yet up to the task. First and foremost, the execution times for common spatial operations, either comparison functions or spatial set operations, are not acceptable. The algorithms implemented in CGAL do have an optimal theoretical runtime, but for practical purposes a series of (simple) optimizations should be considered. Faster algorithms for specific cases are also an option to improve the situation. A smarter memory management to reuse previously existing buffers where possible could add to that as well. Today, CGAL already avoids excessive copying of Nef-polyhedra by adopting a sharing concept, but that can be further improved when it comes to the direct memory management, i. e. when completely new objects are created. It goes hand in hand with a slow and apparently inefficient approach to serialize Nef-polyhedra or to construct a Nef-polyhedron from its serialized representation.

We encountered another major issue during the development of the 3D Extender, namely the coding style of the CGAL library itself. CGAL can actually not be used as a library despite its name. It contains a substantial amount of assertions which will instantly abort the currently running program if violated. For routines executed in a database context, that will lead to an abend of the database system process. It may not by tolerable as it can potentially bring down the complete database system, even if the DBMS took precautions. Omitting the assertions as is explained in [CGA02] is not an

option either as the 3D Extender receives user input, which must be validated before it can be processed safely. Propagating an error code or exceptions through CGAL are better choices for program development [Bal00].

From the perspective of a developer who is not intimately familiar with the CGAL code, the excessive use of templates for the generic programming amounts to a problem once the project reaches a certain size and complexity. Debugging such code is complicated because it is harder to track down specific information in the data structures. The adoption of a suitable object-oriented design to plug in new logic or data types should give an improvement in the maintainability.

5.6 Summary

The support for three-dimensional data in a spatial database system is not yet integrated into the SQL/MM spatial standard [ISO03d]. The first steps to add the facilities for Z and M coordinates to points, linestrings, and polygons were introduced in the current working draft of the standard. Thus, the need for higher-dimensional data has been noticed by the standardization committee. However, at the current stage it is not yet possible to truly model and store polyhedra or even more generalized solids in a spatial database or to perform spatial operations in $\mathbb{R}^3$.

In the current chapter, we described a seamless extension to the spatial type hierarchy along with a set of methods specific for the new data types. An abstract type ST_Solid is added to establish a subtree in the hierarchy dedicated for the geometric primitives for 3D spatial objects. Inheriting the properties of ST_Solid, the purpose of the type ST_Polyhedron is to represent objects with polyhedral surfaces in $\mathbb{R}^3$. Collections of solids and polyhedra are modeled with the types ST_MultiSolid and ST_MultiPolyhedron, mirroring the geometric primitives. The existing spatial data types and their functionality do not need to be modified in any way. We did not discuss the handling of three-dimensional objects with curved surfaces. This omission can be justified by the fact that todays available spatial extensions do not even implement the already standardized data type ST_CircularString to provide the most basic curves: circles and circular arcs. We deemed the introduction of curves at the higher and more complex level of 3D objects to be too excessive for the time being.

In order to load polyhedra and multi-polyhedra into the spatial database or to extract them from there, we extended the standardized external formats well-known text and well-known binary. The extension followed the spirit of the current approach by defining higher-dimensional, more complex objects based on the definitions of existing, lesser-dimensional objects. Thus, polyhedra are constructed from collections of polygons in three-dimensional space. The Geography Markup Language (GML) [OGC04] was not considered as it is just another textual format with a very similar structure as the WKT

representation. Furthermore, GML already mentions some facilities to describe solid objects, even though these facilities are not specified in any further detail.

We implemented the theoretical concepts in a 3D Extender that was based on the existing DB2 Spatial Extender. We could show that the extension of the type hierarchy can be easily accomplished. The differentiation between the original operations in $\mathbb{R}^2$ and the new operations in $\mathbb{R}^3$ was based on the SRS associated with each spatial datum. The 3D Extender provides a set of representative methods for each group of spatial functionality, i. e. conversion routines between the internal representation and external data formats, spatial comparison functions that can be used in predicates for spatial joins, spatial set operations and other methods that generate new geometries, and methods that extract spatial properties from a given geometry.

Summarized, the 3D Extender has demonstrated the validity and feasibility of the theoretical concepts. However, in order to reach a version of the 3D Extender with product quality, additional work has to be invested, in particular to address performance and maintainability concerns. Also, a full integration of three-dimensional data and objects in existing products like PostGIS [Ref05] is desirable as it would avoid the majority of the integration issues and work-arounds adopted by the 3D Extender. Fortunately, the underlying approach does not have to be modified for that work.

Part III

Heterogenous Spatial Database Environments

6 Implementation Aspects of Spatial Data in Distributed Environments

Database environments used in businesses are very often quite complex systems. It is rather rare that just a single database server or the same database management system is installed everywhere. Various groups or teams within an enterprise or corporation have differing requirements for the applications they need to use in order to perform their business tasks. And each group might have its own database system to persistently store the information being processed.

As enterprise-wide integration becomes more and more important today, integrating the data in various database systems of each group is essential. Such a data integration at the relational level can be achieved by different approaches. Providing online access (federation) and copying (replicating) data between the databases is explained in more detail in the Chapters 7 and 8, respectively. Further integration techniques like Enterprise Application Integration (EAI) or Extract/Transform/Load (ETL) were also invented. We do not consider those, however. EAI plays an important role for integration tasks at the application level. We focus at the relational database world instead. ETL tools that are dedicated to the handling of spatial data already exist. An example is the Feature Manipulation Engine (FME) provided by Safe Software [Saf05b]. Therefore, the area of ETL is not further analyzed either.

The topics of replication between homogeneous and heterogenous systems and federated data access are already discussed extensively in the literature [Dad96, CDK05]. However, the specific aspects of spatial data are mostly ignored. Given that spatial information is an integral property of most database systems, spatial data (cf. Chapter 2) should be dealt with as well. An example where data was integrated across various groups is the administration of the city of San Francisco [Int02]. The city uses federated database technology to provide the infrastructure for its 61 municipal departments. Fully supporting spatial data in this scenario would have been very beneficial.

The implementation of the project in San Francisco gave a clear statement that the use of a new product was not an acceptable option in their production systems, much less the use of unsupported prototypical software that is often used as proof-of-concept studies in research. Established and well supported products are a key requirements for institutions and corporations. That implies that the existing products have to deal with the integration of spatial data in such heterogeneous environments. For example,

there is no point to implement a new federated wrapper to access an Oracle database system from a DB2 database system just to enable spatial data access, given that such a wrapper (without the spatial support) already exists [IBM04a]. Developing a new wrapper is not a simple task if production-quality shall be achieved [Lig02] and research prototypes are usually not even close to satisfy such high demands or to provide the full scope of the necessary functionality. The same reasoning is true for replication scenarios. The bottom line is that – for most practical purposes – the means to accomplish the access to distributed spatial data must be based on the currently available, feature-rich and stable but spatially unaware commercial or open source products.

The underlying reasons for the high complexity of access to distributed spatial data are common to federated systems and replication. We demonstrate the widely differing approaches to implement the SQL/MM spatial standard [ISO03d] in Section 6.1. The internal details of the various spatial extensions are explained. Another issue originates from the fact that all spatial information needs to be interpreted with respect to its spatial reference system (SRS). Sections 2.1.2 and 2.3.5 already introduced the concepts of different spatial reference systems in geographic and non-geographic applications and how the SRS information is maintained in a relational spatial database. It is apparent from the SQL/MM spatial standard that no requirements are imposed on the identifying names of spatial reference systems. Consequentially, each spatial product uses different implementations and – even more importantly – different identifiers for SRSs. Therefore, SRSs must be synchronized in distributed database systems and a mapping between SRSs in the various spatial databases becomes necessary. We describe the mechanisms to establish such a mapping in Section 6.2. Both of these implementation-specific topics are summarized in Section 6.3. Based on the findings, the details for federated spatial access and spatial replication are discussed in the subsequent Chapters 7 and 8, respectively.

6.1 Implementation Details of Spatial Extensions

The existence of the SQL/MM spatial standard does not imply that a seamless and portable access to spatial information in different relational database management systems is possible today. Instead, implementations of the standard can come with slight differences that are significant for applications. Additionally, not all spatial extensions adhere to the standard in the first place (cf. Section 2.5). Major variations can be found in the spatial type hierarchy and how it may be used.

The definitions of the spatial data types in the SQL/MM spatial standard are built on structured types. However, the conformance rules in clause 16 of the standard do not imply that structured types need to be used by an actual implementation as long as the various geometry types and the inheritance rules are provided somehow. That omission leaves standard-conforming implementations the maximum leeway for the type definitions. For example, the Informix Spatial DataBlade [IFX02b] employs opaque types and

a series of cast functions to model the subtyping polymorphism. However, the issue to access spatial data in heterogenous database systems goes deeper than just the different ways how the SQL/MM spatial type hierarchy may be implemented. Some systems are still built in a non-object-relational fashion by breaking all geometry information into so-called feature tables, as the OpenGIS Simple Features Specification for SQL (SFS) [OGC99] originally defined as one option. Other systems exploit structured types (also known as object types) but do not provide an explicit type hierarchy. An example for that is Oracle Spatial [Ora05f] where only a single type is used to represent the various kinds of geometries.

Subsequently, we explain the specific implementation of important products. Even with the existence of the SQL/MM spatial standard, it becomes obvious that the specific implementations vary widely, requiring a higher abstraction level when heterogeneous spatial data shall be accessed and managed. Otherwise, each spatial application will have to be tailored to the different implementations.

The following sections go into the details of the implementation of the spatial data types. Additionally some sample code is shown to demonstrate how a polygon with a hole has to be constructed with each product. The same polygon is used everywhere and *Figure 6.1* shows that very geometry.

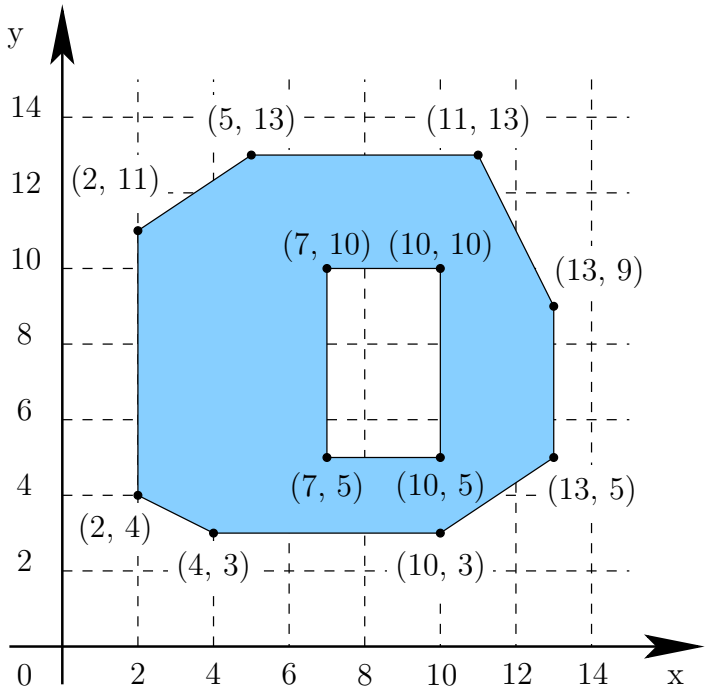


Figure 6.1: Sample polygon with a hole

6.1.1 IBM DB2 Spatial Extender

The DB2 Spatial Extender [IBM04d] implements the SQL/MM spatial standard very closely. It exploits DB2's structured types for the type hierarchy as Section 2.5.1 already explained. The internal implementation of the structured types deviates from the standard as is apparent from *Listing 6.1*. Using DB2's information schema (catalog), the SQL statement to create all the spatial data types can be extracted from a spatially enabled database. Only the type ST_Geometry, the root of the hierarchy, defines any attributes. All other types solely establish the hierarchy and serve as an anchor for type-specific methods like *ST_Length* for curves or *ST_NumRings* for polygons.

```
CREATE TYPE ST_Geometry AS (
      srid        INTEGER,
      numPoints   INTEGER,        geometry_type  SMALLINT,
      xMin        FLOAT,          yMin           FLOAT,
      xMax        FLOAT,          yMax           FLOAT,
      zMin        FLOAT,          zMax           FLOAT,
      mMin        FLOAT,          mMax           FLOAT,
      area        FLOAT,          length         FLOAT,
      anno_text   VARCHAR(256),   points         BLOB(1M),
      ext         SE_Extension )
   NOT INSTANTIABLE  WITHOUT COMPARISONS
   NOT FINAL  MODE DB2SQL  WITH FUNCTION ACCESS
```

Listing 6.1: Type definition for ST_Geometry in the DB2 Spatial Extender

The semantics of the attributes can be derived from their name and also the way how the attributes are used in some of the spatial function. Based on that assessment, a short description of each attribute is given:

srid Each geometry is associated with a spatial reference system (SRS). All SRSs are maintained in a catalog view named ST_SPATIAL_REFERENCE_SYSTEMS. Each SRS can be identified either by its name or the numeric spatial reference system identifier. The srid attribute stores this numeric identifier. It is used to extract the definition of the SRS from the catalog whenever spatial processing takes place.

numPoints This attribute stores how many points are used to define the geometry. For example, if a linestring is given by its well-known text (WKT) representation as "linestring (10 10, 10 20, 20 20)", then the attribute stores the value 3 because three points are contained in the representation.

Internally, the primary purpose of this attribute is to determine whether a geometry is empty or not because no separate data type exists to represent empty geometries.

geometry_type The complex spatial functionality like the computation of intersections is implemented in an external library. The value of the geometry_type attribute is used to communicate between the database server and the library what specific type of geometry is passed to or returned from a spatial routine. The types are encoded in the lower eight bits of the value, and the lowest two bits thereof indicate the presence of M coordinates (measures) and Z coordinates, respectively. *Table 6.1* lists the assignment of type identifiers for the instantiable subtypes of ST_Geometry. There is always a range of four type-identifiers allocated due to the encoding of Z and M coordinates.

Specific geometry type	Type id
ST_Point	04 – 07
ST_MultiPoint	08 – 11
ST_LineString	12 – 15
ST_Polygon	16 – 19
ST_MultiLineString	20 – 23
ST_MultiPolygon	24 – 27

Table 6.1: Values for geometry_type attribute in DB2's ST_Geometry type

bounding box The attributes xMin, xMax, yMin, yMax, zMin, zMax, mMin, and mMax contain the values for the minimum bounding box (MBB) of the geometry. For geometries in $\mathbb{R}^2$ the MBB is the smallest axis-parallel rectangle. This rectangle is often named minimum bounding rectangle (MBR) and it is defined by the minimum and maximum X and Y coordinates of the geometry. The rectangle collapses for points and horizontal or vertical linestrings to the point or linestring itself. The values for the minimum and maximum Z and M coordinates are likewise set if the geometry contains coordinates in either dimension. Otherwise, those attribute values are set to NULL.

The primary purpose of those values is to provide a direct access to the MBR for indexing purposes [Ott06]. However, only the minimum and maximum X and Y coordinates are used for that. Furthermore, those eight attributes provide a fast access to those properties of the geometry.

area and length One-dimensional geometries, i. e. curves and multi-curves, always have a length as an inherent property. Two-dimensional surfaces and multi-surfaces always have an area covered by the geometry. Those information is stored in both attributes for a fast and direct access. Additionally, the attribute length contains the length of the boundary for two-dimensional geometries. The length and the area in both attributes is measured in the units defined for the spatial reference system of the geometry.

anno_text A textual annotation can be stored for each geometry in this attribute. However, there are no routines that read or set the attribute. Furthermore, such annotations usually belong into a separate column in a relational table so that the existence of the attribute in the type is not obvious.

points The actual geometry information can be quite complex. It is stored as a compressed binary large object (BLOB). The compression itself and the encoding of the uncompressed data is not documented, implying that the content of the attribute is completely internal. The internal encoding contains the exact coordinates of the points that define the geometry where the points are already converted to integer numbers using offsets and scale factors of the SRS associated with the geometry.

The BLOB is restricted to a maximum size of 1 MB, and this restriction sets the limit for the number of points. After the compression, each coordinate value takes at least one byte, so that a geometry can have at most 500,000 points in total. A geometry with more points cannot be processed in the DB2 Spatial Extender.

ext This attribute was added to the ST_Geometry type to enable a way to extend the type in the future without breaking backward compatibility or requiring migration of existing spatial data. The declared type of the attribute ext is the structured type SE_Extension. Usually this value is set to NULL, but an application could create a subtype of SE_Extension and hook values of that subtype into each geometry. A possible use for that attribute was explained in Section 5.5.2 where the 1 MB limit of the points attribute quickly amounts to a problem for three-dimensional geometric objects.

Strictly spoken, a user of the DB2 Spatial Extender does not have to know any internals of the types themselves. The values for the attributes are maintained in an object-oriented fashion and the methods operating on the spatial types take care of maintaining a consistent state of a geometry during construction or modification through SQL statements. *Listing 6.2* shows the SQL statement that constructs a polygon with a hole (cf. Figure 6.1) and extracts the area of that polygon. Only the documented and supported functionality, i. e. the constructor *ST_Polygon* and the method *ST_Area*, which is attached to the ST_Surface type, are used. The single 1 (one) that is set in italics in the listing identifies the spatial reference system for the geometry.

```
VALUES ST_Polygon('polygon((2 4, 4 3, 10 3, 13 5, 13 9,
          11 13, 5 13, 2 11, 2 4),(7 5, 7 10, 10 10,
          10 5, 7 5))', 1)..ST_Area()
```

Listing 6.2: Example to work with DB2's spatial types

6.1.2 IBM Informix Spatial DataBlade

The Informix Spatial DataBlade [IFX02b] also follows the SQL/MM spatial standard. It does not use structured types to define the spatial type hierarchy, however. Instead, each type in the hierarchy is implemented as an opaque type. Therefore, it is not possible to derive the internal structure neither from the system catalog tables nor the manual as such information is not documented. Thus, the Informix Spatial DataBlade follows very closely the object-oriented principle of information hiding and the user has no access to any of the internals.

Relationships like inheritance cannot be specified for opaque types. To close this gap, a series of implicit cast functions is defined to provide an automatic substitution of values of a subtype if a value of a direct or indirect supertype is expected in the respective context. For example, an implicit cast is used when a value of type ST_Point is passed to the function *ST_IsSimple*, which operates on all ST_Geometry values.

Type hierarchies built with standardized structured types would also support the `TREAT` operator for down-casting. Given that Informix Dynamic Server [IFX04a] does not provide such an operator, it had to be implemented in a different way. The DataBlade module defines another set of explicit cast operators for that purpose. Combining the explicit with the implicit casts establishes the same behavior as traditional type hierarchies functionality-wise. Extending the hierarchy – as can by done with standardized structured types – is not as straight-forward. The definition of implicit and explicit cast functions is required to handle the new types transparently.

Internally, each geometry stores a numeric identifier for the spatial reference system associated with the geometry. The SRS information itself is maintained in the catalog table SPATIAL_REFERENCES. Spatial functions access this information when it is needed, i. e. when the definition of the SRS is relevant for spatial computations.

Invoking spatial functionality is a straight-forward task. *Listing 6.3* demonstrates how the polygon from Figure 6.1 is created on the fly and the area covered by that polygon is calculated. Upon creation of the geometry, the SRS needs to be associated and that is done in the standardized fashion by providing the numeric identifier of the SRS as an input parameter to the constructor function *ST_PolyFromText*. The SRS identifier 1 (one) is set in italics in the listing for better illustration.

```
EXECUTE FUNCTION ST_Area(ST_PolyFromText('polygon(
   (2 4, 4 3, 10 3, 13 5, 13 9, 11 13, 5 13, 2 11, 2 4),
   (7 5, 7 10, 10 10, 10 5, 7 5))', 1))
```

Listing 6.3: Example to work with Informix' spatial types

The Informix Spatial DataBlade allows that the well-known text representation of a geometry can be directly inserted into a table without explicitly calling a constructor function. An implicit cast function is applied in such a case. In order to handle different SRSs in those situations, a modified version of the WKT representation can be

used where the numeric SRS identifier is placed directly in front of the textual description. *Listing 6.4* shows such an example where the data is inserted into the column GEOMETRY of a table named T. The enhancement of the WKT is not covered by the SQL/MM spatial standard. It should also be noted that other products like PostGIS [Ref05] provide a similar feature but use a different syntax for the SRS identifier.

```
INSERT INTO t(geometry)
VALUES ('1 polygon((2 4, 4 3, 10 3, 13 5, 13 9, 11 13,
         5 13, 2 11, 2 4),(7 5, 7 10, 10 10, 10 5, 7 5))')
```

Listing 6.4: Using the modified WKT representation in Informix

6.1.3 Oracle Spatial

Oracle Spatial [Ora05f] follows a different approach for modeling and processing spatial data in an Oracle database system as Section 2.5.5 already stated. Like PostGIS, no type hierarchy (as mandated by the SQL/MM spatial standard) was implemented. Only a single data type named SDO_Geometry is available. It is an object type, which is comparable to a structured type in [ISO03i]. Its definition is shown in *Listing 6.5*. Differing from PostGIS, the user has to be aware of the internal structure of the type in order to work with such geometries.

```
CREATE TYPE SDO_Geometry AS OBJECT (
   sdo_gtype       NUMBER,
   sdo_srid        NUMBER,
   sdo_point       SDO_POINT_TYPE,
   sdo_elem_info   SDO_ELEM_INFO_ARRAY,
   sdo_ordinates   SDO_ORDINATE_ARRAY )
```

Listing 6.5: Type definition of SDO_Geometry in Oracle Spatial

There is no hidden, internal representation of a geometry as the other products choose to do. All the relevant coordinate information is stored explicitly in the attributes.

sdo_gtype The type of the geometry is encoded in this attribute along with additional properties that are needed to correctly interpret the coordinate array in the sdo_ordinates attribute. The values for the sdo_gtype are comprised of four digits using the format *dltt*. The position *d* encodes the dimensionality of the data space (2, 3 or 4). The value *l* identifies the dimension of the M coordinates (measures) of a geometry in a three-dimensional linear referencing system. It can be either 3 or 4 unless the geometry does not contain any measures, in which case 0 (zero) is to be used. The last two digits *tt* actually encode the geometry type itself according to *Table 6.2*.

Geometric object	*tt* value
point	01
linestring or circularstring	02
polygon	03
heterogeneous collection	04
multi-point	05
multi-linestring or multi-circularstring	06
multi-polygon	07

Table 6.2: Values for geometry type identifiers in the sdo_gtype attribute

sdo_srid The identifier of the SRS that is associated with the geometry is maintained in this attribute. The catalog view SDO_COORD_REF_SYS stores information about all spatial reference systems available in the database. The specific SRS can be identified by the value in the sdo_srid attribute.

sdo_point Representing a point geometry does not require an array being defined. Instead, a single set of X and Y coordinates, along with an optional Z coordinate is sufficient. The attribute sdo_point exists in order to optimize this special but highly important case. The data type for the attribute is itself an object type, and its definition is presented in *Listing 6.6*.

```
CREATE TYPE SDO_Point_Type AS OBJECT (
   x NUMBER,     y NUMBER,     z NUMBER )
```

Listing 6.6: Type definition of SDO_Point_Type in Oracle Spatial

The geometry type in sdo_gtype specifies if the geometry is just a single point. If the attributes sdo_elem_info and sdo_ordinates are both NULL, the attribute sdo_point is consulted.

sdo_elem_info and sdo_ordinates The coordinates of the points that define the geometry have to be stored in the sdo_ordinates attribute. The information in sdo_elem_info structures the point data. Both attributes are actually arrays of numerical values. The size of the arrays is limited to 1,048,576 elements. That implies that a geometry can be defined by at most 524,288 points if only X and Y coordinates are used.

Each part of a geometry is described in sdo_elem_info by a triplet. The first value stands for the index into the sdo_ordinates array where the first point of the part is stored. The second value identifies the type of the part, e. g. outer ring of a polygon or next part of a multi-linestring. And the third element indicates how the single points in the geometry definition are to be connected. The connection can be done using line segments or circular arcs. Thus, a high flexibility is achieved with this data structure.

The major difference to all other products is that an application working with Oracle Spatial has to construct the spatial values itself and ensure that the attributes are filled consistently. Release 10*g* introduced constructor functions like *from_WktGeometry* and *from_WkbGeometry* that are very similar to the two SQL/MM spatial functions *ST_GeometryFromText* and *ST_GeometryFromWKB* and, thus, provide a simpler interface that also eliminates the mentioned pitfalls.

An example how the geometry from Figure 6.1 can be defined in Oracle Spatial is given in *Listing 6.7*. The first three parameters specify that the geometry is a polygon with only X and Y coordinates, has no associated SRS and the attribute sdo_point is not used. Next comes an array giving the information how to interpret the array of coordinates. The first 1 (one) states that the first ring of the polygon begins with the first coordinate. The following 1003 identifies the ring as outer ring, and the final 1 in the triplet stands for linear instead of circular connections between the points of the ring. Likewise, the second triplet represents the inner ring (identified by 2003), which starts at the 19th position in the ordinates array. The coordinate pair of the first point of the second ring is set in italics. The inner rings also uses linear interpolation between its points.

```
SDO_Geometry ( 2003, NULL, NULL,
   SDO_ELEM_INFO_ARRAY(1, 1003, 1, 19, 2003, 1),
   SDO_ORDINATE_ARRAY(2, 4, 4, 3, 10, 3, 13, 5, 13, 9,
      11, 13, 5, 13, 2, 11, 2, 4, 7, 5, 7, 10,
      10, 10, 10, 5, 7, 5) )
```

Listing 6.7: Example to work with Oracle Spatial types

6.1.4 PostgreSQL & PostGIS

The spatial extension for the open source PostgreSQL database system [PSQ05] is named PostGIS [Ref05]. It can be considered as a combination of the implementation of the DB2 Spatial Extender and Oracle Spatial. It follows the SFS in terms of supported functions but it does not provide the type hierarchy of [OGC99] or [ISO03d]. Just a single geometry type Geometry exists. That type can be used to model the geometric primitives points, lines or polygons. Homogeneous and heterogeneous collections can also be handled with this type.

The type Geometry is an opaque type. Internally, it stores the geometry information in a way that is very similar to the well-known binary format. Due to the availability of the source code, the internal structure can be analyzed directly. *Listing 6.8* shows the definition of the format using a Backus-Naur Form (BNF) for illustration.

<postgis geometry encoding> ::=
    <type id> [*<bounding box>*] [*<srs id>*] *<coordinates>*

Listing 6.8: Type definition for Geometry in the PostGIS

Contrary to the well-known binary (WKB) format, no endianess information [Coh81] is required as the geometry is always stored with the endianess of the underlying hardware platform. The other attributes have the following meanings.

type id The type identification consists of four bytes and it has a structure of its own. The lowest eight bits are used for the actual type identifications as defined for the WKB representation. The remaining bits are used to flag the existence of Z and/or M coordinates, the optional bounding box, and the optional SRS identifier of the geometry.

bounding box If present, the bounding box contains the minimum and maximum X and Y coordinates. Only four byte floating point values are used for each in order to reduce the space requirements of a geometry. The space savings are traded in for a reduced precision, of course.

srs id A geometry in PostGIS may have an associated SRS. If that is the case, then the existence of the SRS identifier is flagged in the <*type id*> value. The actual SRS definition is stored in the table SPATIAL_REF_SYS and the geometry only contains the respective numeric identifier.

coordinates The coordinates are stored after the header exactly as the WKB format prescribes, i. e. each connected set of points (like a ring) consists of a counter indicating the number of points, followed by the X and Y coordinates as well as Z and M coordinates (if applicable) for each point. Likewise, sets of parts of geometries (e. g. multiple rings) or sets of geometries (e. g. geometry collections) carry such a counter to specify how many elements the respective set contains.

Like the opaque types employed by the Informix Spatial DataBlade, the user of PostGIS does not have any knowledge about the internal storage structure for the spatial data. The management of spatial data has to be performed solely by using the functions provided by PostGIS. *Listing 6.9* demonstrates how the polygon from Figure 6.1 is to be constructed and a function applied to that polygon. Because no SRS is necessary, it is not given in the example. Due to the adherence to the SFS, the usage of PostGIS is intuitive if other spatial products are already known.

```
SELECT Area(PolyFromText('polygon((2 4, 4 3, 10 3, 13 5,
          13 9, 11 13, 5 13, 2 11, 2 4),(7 5, 7 10, 10 10,
          10 5, 7 5))'))
```

Listing 6.9: Example to work with the PostGIS spatial type

Similar to the Informix Spatial DataBlade, PostGIS addresses the problem that the standardized WKT and WKB representations do not carry any information about the SRS associated with a geometry. Therefore, at least the WKT format was extended by

prepending the SRS identifier as in *Listing 6.10*. The SRS identifier in the extended WKT is set in italics. The chosen syntax is actually different from the one adopted by the Informix Spatial DataBlade.

```
INSERT INTO t(geometry)
VALUES ( GeomFromEWKT('SRID=1;polygon((2 4, 4 3, 10 3,
            13 5, 13 9, 11 13, 5 13, 2 11, 2 4),
            (7 5, 7 10, 10 10, 10 5, 7 5))') )
```

Listing 6.10: Using the extended WKT representation in PostGIS

No issues regarding inheritance could arise in PostGIS because the type hierarchy is omitted and strong typing is not provided. Thus, the implementation of PostGIS became much simpler in this respect.

6.2 Management of Spatial Reference Systems

To handle spatial reference systems (SRSs)[1] is often neglected by spatial applications. Even some of the spatial extensions for relational database systems ignore it [MyS05]. However, each coordinate value is only well-known with respect to its SRS. If the SRS is not maintained in the spatial extension, then the application must take care of it itself and interpret the coordinates properly.

Many different SRSs are already defined by various organizations, the majority of which is specifically dedicated to geographic data processing. For example, the European Petrol Survey Group (EPSG) provides a database with more than 2800 different SRSs [EPS04]. There are manifold reasons for such a wide variety. The technology to measure and calculate the center of the Earth has improved significantly during the last two centuries. Thus, a spheroid that approximates the shape of the Earth could be determined with a much higher accuracy. For example, the spheroid commonly used for coordinates with latitude/longitude values, the World Geodetic System (WGS), was defined in 1960 and then refined in 1964, 1972 and again in 1984. A second reason is that many SRSs are developed for specific areas. Those SRSs may be geographic or projected coordinate systems (cf. Section 2.1.2). For example, using the SRS WGS84 for data in the highland of Tibet would introduce inaccuracies due to the elevation of the area of interest.

Depending on the source of the spatial data, e. g. sensors, actuators, users, or other devices like the Global Positioning System (GPS) receivers, the spatial data may be available in many different SRSs [SHN04]. Converting all the data into a single, global SRS is – although possible – usually not an option. Data loss or reduced accuracy must be prevented, or auditing requirements may even forbid conversions. Furthermore, in the case of distributed spatial databases, the spatial data is already available in a

[1] Spatial reference systems are also known in the literature as *coordinate systems*.

given SRS and it has to be dealt with that way. Given the very wide variety of spatial reference systems, it is essential to cope with different SRSs. That is even more true for distributed spatial database systems where an application requires a consistent and integrated view on the spatial and non-spatial data. Data originating from different databases will be processed together, for example spatial joins may be performed or the integrated data is to be rendered in a single view.

Two geometries can only be compared, processed or visualized together if both are represented in the same SRS. Therefore, the following operations on SRSs are mandatory for distributed spatial databases:

1. comparing different SRSs and testing for their semantic equality, and
2. implicitly and explicitly transforming geometries from one SRS to another.

Spatial reference systems are maintained in several locations in distributed environments. To be more precise, each spatial database maintains its own SRSs and each assigns numeric identifiers to them independently. Thus, it is possible that a given SRS srs_a may have the identifier 5 in one database and the identifier 8 in a second database. Likewise, an SRS srs_b can exist in the second database and carry the identifier 5. The bottom line is that SRSs cannot be compared solely based on their identifiers in the various spatial databases. A mapping between SRSs in different spatial databases has to be based on their actual definition. Once such a mapping is established, the access to distributed spatial data can be achieved by simply replacing the numeric identifier of the SRS used in the source database with the identifier of the matching SRS in the target database. Source and target databases are obvious in replication configurations. In federated environments (cf. Chapter 7), the source database is the foreign data source and the federated server represents the target database.

The matching of SRSs in the source and target databases shall not be done each time when a spatial value is communicated between both spatial databases. Therefore, the SRS mapping is treated like any other meta data and stored in the information schema of the replication tool or the global schema of the target database at the federated server, for example. Meta data management systems [Kim05] could also be exploited for that task. The exact storage mechanisms depend on the access method to the distributed spatial data, e. g. federation or replication. We explain the necessary enhancements in more detail in the subsequent Chapters 7 and 8. That also includes mechanisms to maintain the mapping in case of a modification of an SRS definition in either the source or target database.

Matching two spatial reference systems is an easier task if the spatial extension adheres to the SQL/MM spatial standard and uses the standardized well-known text representation for SRSs. We explain in Section 6.2.1 how the comparison can then be accomplished. Section 6.2.2 describes a more general mechanism that is based on a canonical, normalized representation for SRS.

6.2.1 Direct Mapping

The spatial products available for the three major database systems DB2 UDB, Informix, and Oracle Database all provide mechanisms to define or extract an SRS based on its WKT representation. The names of the respective tables or views of the spatial information schema vary, but the information is available. Thus, it is possibly to use a direct route to align the SRSs in those systems by using the method *ST_Equals* that is defined in the SQL/MM spatial standard for the type ST_SpatialRefSys. Values of this type model the complete SRS definition with all its information about ellipsoid, prime meridian, units of measure, etc. The method *ST_Equals* compares the single elements and returns 1 (one) if the current SRS (subject parameter) is identical to the other. A simple textual comparison of the WKT is not an option because leading or trailing zeros in numerical values, different spacing, and different ordering of the elements in the WKT may occur and lead to mismatches for otherwise identical SRSs.

An obstacle with this direct comparison is the fact that the DB2 Spatial Extender [IBM04d] and the Informix Spatial DataBlade [IFX02b] both require that an SRS carries additionally so-called *offsets* and *scale factors* for each dimension. The offsets and scale factors are needed to convert external floating point coordinate values to internal integer values. Integers are used internally to ensure computationally reliable results by reducing or preventing rounding issues during spatial computations. The offset is subtracted from the floating point value and then multiplied by the scale factor. The integer part of the result becomes the internal representation of the coordinate.

Oracle Spatial [Ora05f] does not impose such a requirement to provide offsets and scale factors for an SRS. However, it is recommended to assign a so-called *resolution* parameter to a spatial column [Ora03]. The resolution serves exactly the same purpose as the scale factor in the DB2 Spatial Extender. The floating point values for the coordinates, which are stored in the SDO_Ordinates attribute of the type SDO_Geometry, are multiplied with the resolution. The integer part of the result becomes the representation of the coordinate value that is used during spatial calculations. Oracle Spatial determines a resolution internally during an operation if no explicit resolution was assigned to a spatial column or if the geometry does not originate from a spatial column, e. g. it was provided as a parameter in the SQL statement by the application. The internal resolution is not available to an application (e. g. federated server or replication) that wishes to access the distributed spatial data. Therefore, the resolution has to be specified manually if it is not already available.

An automatic SRS matching can be implemented with the resolution or scale factor defined. Two SRSs are considered to be matching if the following conditions are satisfied:

- the result of the *ST_Equals* method returns 1 (one), and
- the offsets and scale factor at the source system are equal to the offsets and scale factors at the target system.

A more fine-grained comparison of offsets and scale factors is only possible if additional information like the minimum and maximum possible coordinate values is available. That is the case for many geographic coordinate systems, which represent longitude values (X coordinate) in the range $[-180, +180]$ and latitude values (Y coordinate) in $[-90, +90]$. For those cases, two SRSs match if the following condition is fulfilled for the scale factors s_s and s_t at the source and target databases, respectively, and the offset o_t at the target. The value $2^{31} - 1$ is the largest internal integer coordinate value and it may vary depending on the actual target system.

$$(s_s \leq s_t) \wedge \left(+180 \leq 2^{31} - 1 + o_t\right)$$

If there is an SRS in the source database for which no matching SRS can be found in the target database, a new one has to be created at the target. The new SRS should use exactly the same WKT representation, offsets and scale factors as the corresponding SRS at the source to guarantee a match.

6.2.2 Mapping Using a Canonical Representation

A more general approach for the matching and mapping of SRSs is required if either the source or the target database system do not allow for the representation of an SRS in its WKT format. That implies that the spatial extension does not follow the SQL/MM spatial standard and that the type ST_SpatialRefSys and the method *ST_Equals* (or comparable functionality) are not available. In that case, a canonical representation of an SRS is needed. All SRSs to be compared have to be mapped to this canonical format and the comparison is then performed with this format.

The handling of SRSs is an extensively researched field in the industry, most prominently for businesses involved in geological and geographical operations. The European Petrol Survey Group (EPSG) is an organization dedicated to the definition and management of spatial reference systems. It specified a complex schema comprised of 15 relational tables, modeling all possible aspects of an SRS in a semi-normalized fashion [EPS04]. *Figure 6.2* shows the SRS-specific part of the EPSG schema, omitting all irrelevant tables.

The EPSG schema is a true superset of the SRS specifications in the SQL/MM spatial standard. It is possible to store all elements of a projected, geographic or geocentric coordinate system (that is given in its WKT representation) in the EPSG tables. Likewise, the respective information can be extracted from those tables to compose a valid WKT definition for an SRS that is understood by an implementation of the SQL/MM spatial standard. Summarized, it is possible to use the EPSG tables as a canonical representation to compare and match SRSs. We explain how the canonical format can be used.

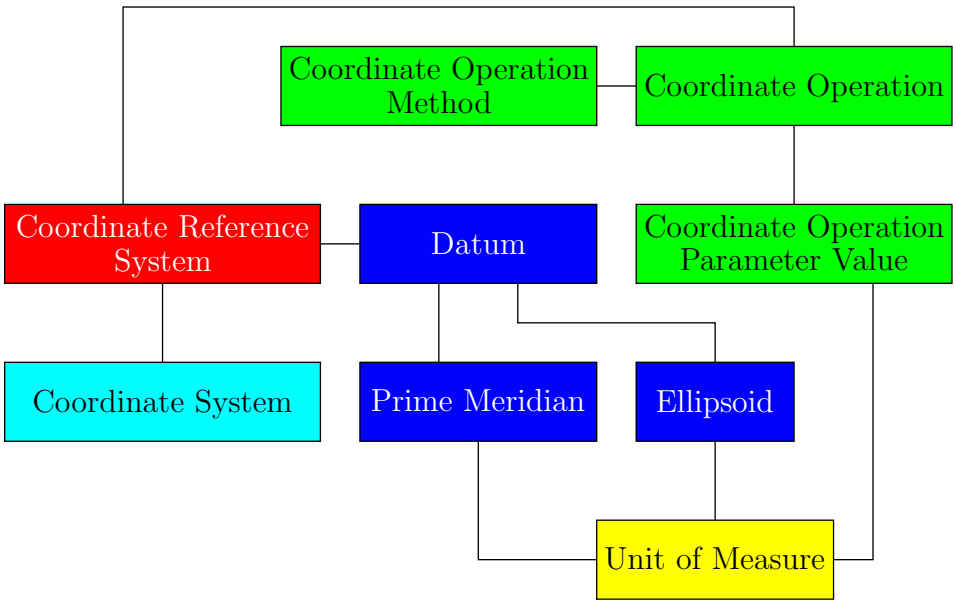


Figure 6.2: EPSG schema for spatial reference systems

Geographic Coordinate Systems

The syntax for a geographic coordinate system is shown in *Listing 6.11* using BNF notation. This definition was originally specified in [OGC99] and is now standardized in the SQL/MM spatial standard. The productions for some of the symbols are not shown if they are either straight-forward or not relevant for the mapping to the EPSG schema.

```
<geographic cs> ::= GEOGCS <left delimiter>
    <double quote> <name> <double quote> <comma>
    <datum> <comma> <prime meridian> <comma> <angular unit>
    [ <linear unit> ] <right delimiter>

<datum> ::= DATUM <left delimiter>
    <double quote> <datum name> <double quote> <comma>
    <spheroid> <right delimiter>

<spheroid> ::= SPHEROID <left delimiter>
    <double quote> <spheroid name> <double quote> <comma>
    <semi−major axis> <comma> <inverse flattening> <right delimiter>

<prime meridian> ::= PRIMEM <left delimiter>
    <double quote> <prime meridian name> <double quote> <comma>
    <longitude> <right delimiter>
```

Listing 6.11: Syntax definition for geographic coordinate systems

The mapping of the information represented by the various non-terminal symbols in Listing 6.11 is summarized in *Table 6.3*. The names of the tables mentioned refer to Figure 6.2. A direct, name-based mapping to the columns in the EPSG schemata exists so that no further considerations on the target columns are necessary. The only additional information is that a geographic coordinate system is *ellipsoidal*, and that is stored via a relationship to the table Coordinate System.

Symbol in BNF	Matching EPSG table
<*name*>	Coordinate Reference System
<*prime meridian name*>	Prime Meridian
<*longitude*>	Prime Meridian
<*angular unit*>	Unit of Measure
<*linear unit*>	Unit of Measure
<*datum name*>	Datum
<*spheroid name*>	Ellipsoid
<*semi−major axis*>	Ellipsoid
<*inverse flattening*>	Ellipsoid

Table 6.3: Mapping SRS WKT to EPSG tables

Geocentric Coordinate Systems

The mapping for geocentric coordinate systems is identical to geographic coordinate systems. That is apparent from the very similar syntax specification shown in *Listing 6.12*. The symbols <*datum*> and <*prime meridian*> are resolved in exactly the same way as for a geographic SRS. Therefore, the rules from Table 6.3 are directly applicable. The only difference is that a geocentric coordinate system is a *Cartesian* SRS and, thus, it has to be reflected that way with another relationship to the Coordinate System table.

<*geocentric cs*> ::= GEOGCS <*left delimiter*>
<*double quote*> <*name*> <*double quote*> <*comma*>
<*datum*> <*comma*> <*prime meridian*> <*comma*> <*linear unit*>
<*right delimiter*>

Listing 6.12: Syntax definition for geocentric coordinate systems

Projected Coordinate Systems

According to the SQL/MM spatial standard, a projected coordinate system is always based on a geographic coordinate system. That is also apparent from the non-terminal symbol <*geographic cs*> in *Listing 6.13*. Therefore, the rules for geographic coordinate systems are used. That leaves the mapping of the projection-specific options.

```
<projected cs> ::= PROJCS <left delimiter>
    <double quote> <name> <double quote> <comma>
    <geographic cs> <comma> <projection> <comma>
    { <parameter> <comma> }... <linear unit> <right delimiter>

<projection> ::= PROJECTION <left delimiter>
    <double quote> <projection name> <double quote> <right delimiter>

<parameter> ::= PARAMETER <left delimiter>
    <double quote> <parameter name> <double quote> <comma> <value>
    <right delimiter>
```

Listing 6.13: Syntax definition for projected coordinate systems

A projection is an operation on coordinates, more particularly, it is a coordinate transformation. Therefore, the table Coordinate Operation associates the *<projection name>* with its actual projection formula. The name of the projection is kept in the EPSG table Coordinate Operation Method. The parameters for the projection are comprised of their *<parameter name>* and the associated *<value>*. Both is stored in the EPSG table Coordinate Operation Parameter Value.

Summarized, all the information from an SRS based on the definitions in the SQL/MM spatial standard can be mapped to the EPSG schema. Likewise, an SRS stored in the EPSG tables can be extracted and its corresponding WKT can be built. If such a forward and backward mapping can be accomplished for other systems that deviate from the SQL/MM spatial standard, then the final matching between two SRSs can be implemented with the EPSG schema as canonical format.

6.3 Summary

The implementations of spatial extensions for relational database systems are extremely diverse. The supported types and routines differ. Sometimes the internal type definitions are encapsulated and other products require the direct manipulation of attributes of the spatial types.

The spatial extensions implement varying enhancements for external data formats. For example, some products allow the specification of SRS identifiers in the WKT. Unfortunately, each of those product uses another syntax. The handling of spatial reference systems is very insufficient in any case because no consistent and standardized numbering scheme across products is in place. Each spatial extension keeps track of SRS on its own.

The overall situation has a significant impact on portable spatial applications. Each application has to be tailored specifically to the underlying spatial extension and RDBMS.

The implication is that the exchange of spatial data between different spatial database systems can only be accomplished via external data exchange formats like WKT or WKB if a high degree of independence and portability shall be achieved. That has a direct impact on spatial federation (cf. Chapter 7) and spatial replication (cf. Chapter 8). A mapping of SRS between components in a distributed system becomes necessary. Such a mapping can either be accomplished directly with the standardized well-known text representation for SRS or a canonical format like the EPSG schema. The results of the mapping process can be stored depending on the needs of the federation or replication processes. A relational storage in an information schema is just one approach.

7 Federated Spatial Databases

A federated database system is a distributed database system that uses synchronous operations to communicate between at least two database systems. A federated system consists of a database server acting as a federated server and one or more data sources[1] that are external to the federated server. The data sources themselves can be relational database systems or other, non-relational storage mechanisms [Con97]. The federated server represents data at the data sources as relational tables to its clients. The distributed and local data can be handled together in a single transaction, even in a single SQL statement. Thus, it is possible to update data at one data source based on data stored at other data sources or at the federated server. Likewise, queries can span multiple data sources and join tables at any of the data sources – across all data sources – and also with the federated server itself.

A client application communicates only with the federated server and may transparently access remote data sources. The external schema presented to the database client consists only of a set of tables. The physical storage of the tables is managed at the conceptual and internal layers of the federated server. The information schema is a global schema. The general architecture of a federated database system is depicted in *Figure 7.1*. The SQL statements sent from the client to the federated server are analyzed and the portions referring to a specific data source are determined and passed on to the respective data source. The results from all data sources and the execution of the portion local to the federated server are collected and combined to be sent as response to the client.

All major database systems provide federated capabilities, although different names were adopted for identical or similar concepts. Even the open source systems MySQL [MyS05] – otherwise rather weak in supporting sound and strong relational concepts – implemented a storage engine that is able to access data in a different, potentially remote MySQL system.

There are many spatial scenarios where federated capabilities are used. A typical situation can be found in companies or in the public sector where different groups developed and operated their database-driven applications independently in the past. Integrating all those environments was often not considered but it became more important during

[1] The SQL/MED standard [ISO03l] calls a data source *foreign server* to indicate that it is not part of the federated server. The federated server itself is referred to as *SQL-environment* consistently throughout all parts of ISO 9075.

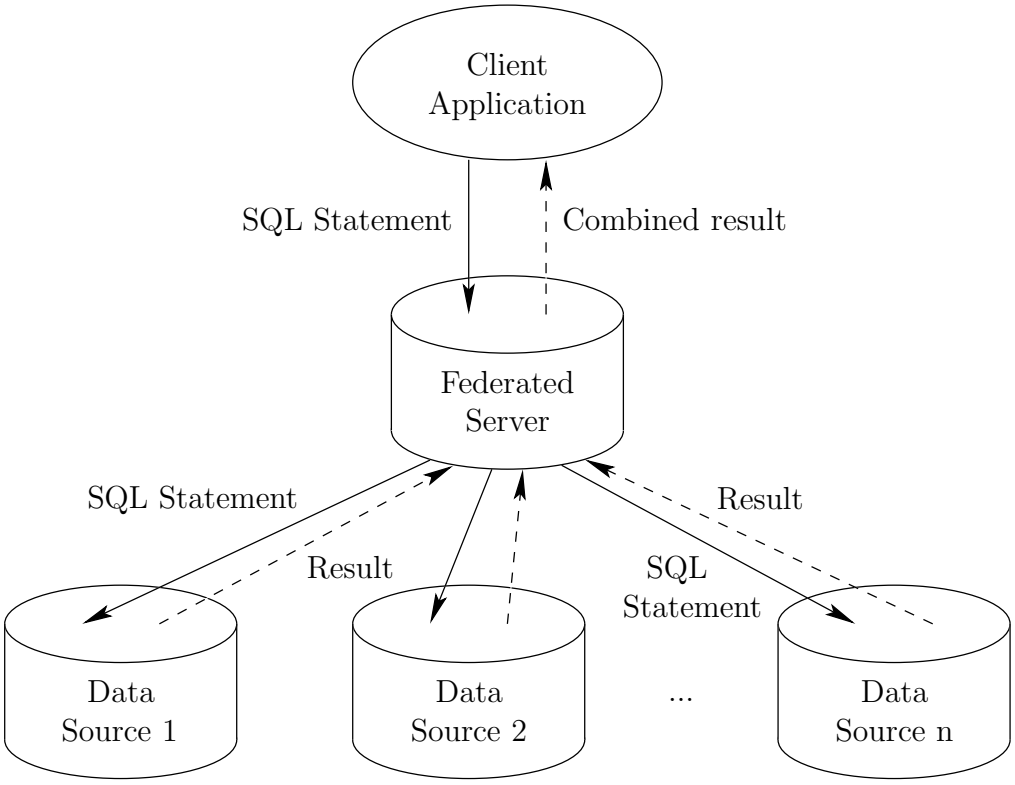


Figure 7.1: Architecture of federated database systems

the recent years. The requirement to integrate the different database systems received also more attention. Migrating all data into a single database system appears to be an initial answer to satisfy such an integration request, but typically that is not a viable option. Migrating whole environments is a very expensive undertaking and sometimes it may not be possible at all for technical reasons, e. g. if applications require a certain platform. Other obstacles for such a migration can have legal or even political backgrounds. Federation offers a solution by allowing the data to be shared without forcing any department or group to give up their control over the data repository they are maintaining. Furthermore, all departments will have access to the most recent information from other groups so that issues with out-of-date information or latencies do not appear.

Federated technology in an enterprise environment can also become necessary if certain data stems from specific sources, for example, highly specialized applications or machines may collect sensor-based data and provide this information via predefined interfaces. Providing relational access to the latest sensor information can be accomplished by registering the sensors as data sources in a federated database system. Alternatives like constantly importing the sensor data have the potential to also satisfy the requirements but it is also possible that they add too much load onto the system, which may not be desired.

A federated system can also be very beneficial if a complex application consists of many different components and stores a vast amount of information in the database. An example would by SAP R/3 [HFK06]. Analyzing the data in the database system, one may find data that is specific to a particular component but also data that is common to multiple or even all components. The common data is a candidate to be sourced-out into a separate database whereas the component-specific information is maintained in dedicated databases. Thus, specific tuning and caching techniques can be implemented. The separation may also be applicable to different customers. The common data can be shared and made available to all databases via federated access.

Another example are replication tools in heterogeneous environments where federated capabilities are exploited. Federation already handles the mapping between the different structures and data types of the involved database systems. Therefore, replication tools only need to take care of the actual replication between identical database management system, circumventing potential problems inherent to heterogenous environments.

The previous examples are by no means a complete list of all possible usage scenarios for federated database systems. Their intent is merely to illustrate various practical situations where federation is used today. It is also obvious that including spatial data in the scenarios is very desirable or even mandatory, especially if spatial data is a primary asset. In the following section we introduce the capabilities defined in the SQL/MED standard [ISO03l] for handling of complex data types like the spatial types. Although the standard takes care of user-defined data types, the currently available federated products fall short of the standard as Section 7.2 shows. Today's applications are built on products and not on standards. Therefore, we document in Section 7.3 how a federated wrapper can be implemented to access a data source that contains spatial data – a geographic information system built on GRASS – and also how the spatial data itself can be accessed. We built a GRASS wrapper for the DB2 database system because DB2 adheres very closely to the SQL/MED standard. We apply the findings from the GRASS wrapper to productized wrappers like the DB2 (DRDA) wrapper. The DRDA wrapper is provided by IBM WebSphere Information Integrator [IBM04a] that accesses a foreign DB2 data source. Section 7.4 explains the adjustments that are necessary not only to access spatial data at the foreign DB2 data source but also how to push down spatial predicates. We close this chapter with a summary in Section 7.5.

7.1 Provisions in SQL/MED

Accessing foreign data sources requires several pieces of information to be known to the federated server. First, we must know the library that implements the remote access. Then, we register the data source itself. Finally, the federated server has to be aware of the single foreign (relational or non-relational) tables at the data source. The SQL standard for the management of external data (SQL/MED) [ISO03l] defines the

functionality for the respective components. Additionally, the standard takes care of authorization questions and it defines how a federated query is to be executed, i. e. how the federated server communicates with the library. *Figure 7.2* shows an overview of all those components, which are briefly introduced subsequently. The components are usually written in a standard programming language like C or Java and collected in a library called the *wrapper*.

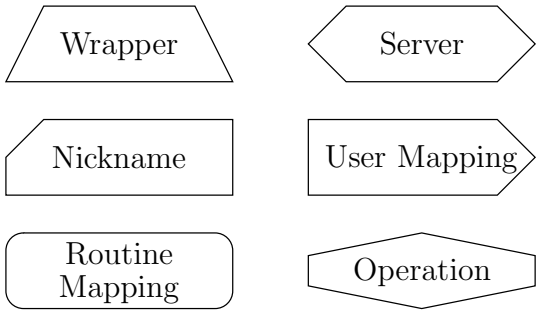


Figure 7.2: Components of a wrapper

Any facilities specific to the handling of spatial information in those components are highlighted. We explain how the spatial data types are to be mapped between the database systems and also how the spatial functionality can be accessed transparently at the federated server.

7.1.1 Foreign Data Wrappers

A foreign data wrapper, or short *wrapper*, is a mechanism to access and to modify data managed in an external system. It defines the communication protocol between the federated server and the remote data source, the foreign server. Thus, a wrapper can be used to talk to different data sources that use the same data management system, for example to several Microsoft SQL Server systems. The specific interface between the wrapper (and foreign server) and the remote system is wrapper-dependent.

At the federated server, a wrapper establishes the entry point for the federated server to communicate with foreign servers and nicknames. The wrapper is directly or indirectly responsible for the initialization and startup of the other components.

A wrapper is registered in a database; information about its version and the owner of the wrapper and the library name are stored in the information schema at the federated server along with other wrapper-specific options. A wrapper can be used to either access relational (SQL-aware) or non-relational (non-SQL-aware) foreign servers.

7.1.2 Foreign Server

A foreign server represents a single data source that is external to the federated server. The data stored at the foreign server is mapped to the relational representation. Every foreign server is accessed through exactly one wrapper, but several servers may share a wrapper. For example, two server objects can be registered at the federated server, and each server represents a single Oracle database, possibly on different machines. If the data source is a relational system, a foreign server usually maps to a database.

The main task of the server component comes into play during query compilation. The federated server asks the server component about the capabilities available at the data source. A request/response protocol is used to come up with the final query plan step-by-step. The federated server gives a part of the plan that it would like to be handled by the data source as a request to the server. The server analyzes each element in the plan and responds with a structure that contains all the elements that the data source can truly handle. All other elements are omitted and, thus, the federated server can build the plan iteratively. For example, if a foreign data source cannot cope with aggregations like *SUM* or joins are not supported, then a plan involving both are rejected and only the supported functionality is communicated to the query compiler. The query compiler can either decide to perform the aggregation/join locally, compensating functionality missing from the data source or it can then choose a completely different plan.

The access to nicknames is the second important task of a server – besides the involvement during the query compilation. As part of handling nicknames, the server is responsible to transparently map the data types from the foreign data source to the corresponding data types supported by the federated server.

7.1.3 Nickname

Some sort of collection of data stored at the foreign data source is represented in tabular structure at the federated server. For relational wrappers, a nickname is merely a pointer or reference to the relational table in the remote system. A nickname could also refer to a file in the file system if the structure of the file content is well-defined and known. The specifics of the mapping are dependent on the wrapper and server implementation. A client application accessing and querying the federated server does not need to be aware of the actual storage of the data available through the nickname. The remote data is represented as if it were a base table or view at the federated server itself.

The SQL/MED standard distinguishes between foreign data sources that do provide an information schema analogous to the SQL Information Schema and those that do not. If such an information schema is available, a nickname may be created by importing the remote schema information. Otherwise, the federated server needs to receive the schema of the nickname when the nickname is created.

Whenever a nickname is referenced in a query, a descriptor for the result set is generated. That descriptor contains a field named CURRENT_TRANSFORM_GROUP_FOR_TYPE for each structured type. Along with the data type, it identifies the transform function (cf. Section 3.2) that the federated server shall invoke to convert the presented data to a value of the target type, e. g. a spatial type. Thus, the transform functions are the means to pass values of structured or opaque types between involved systems. The server component also needs to be aware of the mapping of the spatial types between the data source and the federated server. A similar approach is applicable to the passing of spatial data to the data source.

The SQL/MM spatial standard defines three different transform groups for spatial values:

- *ST_WellKnownText*
- *ST_WellKnownBinary*
- *ST_GML*

Section 3.2 already explained that none of the standardized transform functions is a full description of a geometry value since the information about the associated spatial reference system (SRS) is not available. The transform group *ST_GML* describes the coordinates of a geometry and contains the numeric identifier of the SRS. However, the exact definition is not included. Given that the SRS identifiers may differ in the various systems, the numeric identifier is not sufficient. The other two transform groups do not even provide that identifier. The only remaining way would be to include the SRS identifier as an option for the nickname when it is registered. However, that imposes the restriction that only spatial values in the same SRS may be stored in the table or table-like structure at the foreign data source.

Including the complete SRS definition with each geometry is not desirable for performance reasons. Transferring singles point, each with just 21 bytes in the well-known binary (WKB) representation can be dramatically impacted if the SRS adds another 200 bytes (or more) each time.

The full flexibility that would be achieved by the inclusion of the SRS is actually not needed in federated spatial scenarios. The SRS definitions are usually static. A mapping between the SRS identifiers at the foreign data source and the SRS identifiers at the federated servers can be collected once and used subsequently. We described in Section 6.2 how the spatial reference systems can be mapped between two systems. With such a mapping established, the only remaining issue is to define a proper transform group that also handles the SRS identifier, for example PostGIS has an extended WKB representation as described in Section 6.1.4 would be adequate. A spatial value retrieved from the data source can be created in the proper SRS at the federated server.

7.1.4 User Mapping

A foreign data source may require its own authorization. The users and groups that have access to the data source may not line up with the users and groups with access to the federated server. Additionally, the mechanisms to manage the users and groups may differ in both systems, e. g. one system would use Kerberos or LDAP and the other rely on the operating system.

A user mapping specifies the (authorization) credentials C that shall be used at the data source when user U accesses the federated server. In other words, a query executed at the federated server under the authorization of U is executed at the foreign data source by user C.

7.1.5 Routine Mapping

Similar to user mappings, the foreign data source may support various functions, stored procedures and other routines. If one of those foreign routines has the same semantics as a routine at the federated server, both routines can be mapped to each other. Once such a mapping is established, the federated server can decide during the compilation phase of an SQL statement whether the routine is to be executed at the foreign data source or locally at the federated server. The remote execution can be beneficial if the routine is used in predicates of queries to filter the rows to be returned. With the routine already being evaluated at the foreign data source, the amount of data to be transferred to the federated server can already be reduced there.

Implicit routine mappings can be provided by the wrapper itself. For example, functions like *SUBSTR* exist in virtually every relational database management system (RDBMS) and wrappers do not have to be informed about the routine mapping for it. Routine mappings are typically intended for user-defined functions (UDFs).

Type-specific transform groups are applied when spatial values are involved as output parameters for the remote routine. The foreign data source (or the wrapper) will provide the data in a format that adheres to the requirements of the transform group. The federated server takes that data and invokes the transform function on it to generate the value of the proper user-defined type. This behavior covers the case when spatial data is retrieved from the data source.

7.1.6 Remote Operation

A query will be executed once it is compiled into a plan. The query is constructed in the format required by the remote system and then sent to it. The results are subsequently retrieved and mapped to the format expected by the federated server.

The SQL/MED standard does not define any facilities for remote data manipulation operations. However, existing products like the IBM WebSphere Information Integrator [IBM04c] also support DML statements. Their logic is very similar to queries. The statement is sent to the foreign data source along with the values inserted, or updated.

7.1.7 Processing of Federated Queries

When an application sends a query to the federated server and that query involves data stored at a foreign data source, then a complex process is initiated in order to retrieve the desired data from the foreign data management system. All of the before described components have to play together to accomplish the task. The process can clearly be separated in the planning and the execution phases as *Figure 7.3* indicates with the dashed line. We explain the single steps for both phases in more detail.

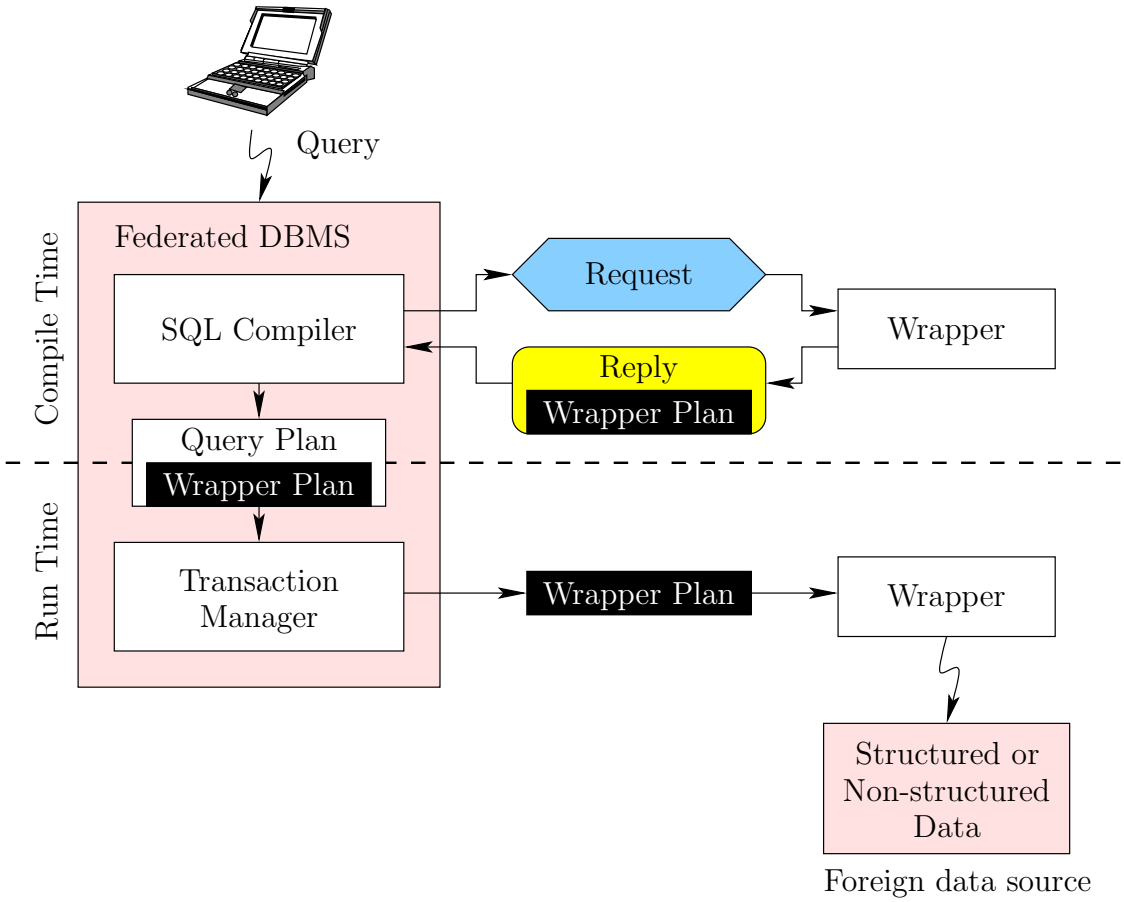


Figure 7.3: Processing a federated query

Query Compilation

1. The client application sends the (federated) query to the database management system (DBMS) that acts as federated server.
2. The federated server determines all relevant wrappers based on the nicknames referenced in the query. The wrapper libraries are loaded and an instance of the wrapper class is created for each. The instances are initialized with the wrapper options.
3. All foreign servers partaking in the query are determined based on the nicknames. An instance of the respective server classes is created via the wrapper objects and subsequently initialized with the server-specific options. For each server:
   a) A request object representing a query fragment related to the foreign data source is sent to the server object. Any relevant routine mappings are already incorporated into the fragment.
   b) The server object analyzes the query fragment and creates zero, one, or more reply objects that contain the accepted pieces of the request. If no reply is returned, the federated server breaks down the request into smaller fragments that are sent again to the server object. If multiple replies are returned, each reply stands for an alternative wrapper plan for the provided request.
   c) The federated server collects a list of predicates that were not accepted by the server object. Those predicates are later evaluated directly at the federated server, i. e. the functionality not supported by the foreign data source is compensated at the federated server.
   d) Selectivity and cost estimates are requested from each server object.
4. The optimizer at the federated server chooses one execution plan for the federated query, which includes one or more wrapper plans.

Query Execution

5. For each server:
   a) Create and initialize a user mapping object with the credentials that shall be used for the authentication with the foreign data source.
   b) Establish a connection for the communication of the wrapper with the foreign data source.
   c) Initialize query execution at the foreign data source by instantiating the remote operation class. The remote operation object uses the opened connection to send the query fragment in the proper format and protocol to the data source.

d) The results from the data source are fetched sequentially. Each fetch transforms the data into a tuple in the format expected by the federated server. This step is repeated until no further results are found at the foreign data source.

e) The remote operation object is cleaned up and destroyed at the end of the query execution.

f) The connection to the data source is terminated at an appropriate point in time, for example when a transaction at the federated server is finished. The user mapping object is removed at this time as well.

6. The server objects are cleaned up and destroyed.

7. The wrapper objects are cleaned up and destroyed.

7.2 Federated Products

The support for spatial data in the federation products is virtually not existing today. As spatial data is usually not implemented in the database server itself but rather as some sort of extension based on user-defined types (UDTs), the basic criteria is the support of such UDTs by the federated technology. When this capability is available at all, it usually ends with the support of distinct types, which are merely a renaming of predefined data types like INTEGER.

In this section, we give short introduction to three database products that offer federated capabilities. The extent for the support of spatial data is analyzed.

Federated scenarios where spatial data is involved (but not the federated access to spatial data) are not discussed. For example, it is possible to access non-spatial data residing at a foreign data source and join it with locally stored spatial data. The vertically partitioned storage of both sets of data can be hidden from the user via views. That is just a traditional federated scenario and well supported today.

7.2.1 IBM DB2 Universal Database

DB2 UDB has built-in federated capabilities. It implements a query gateway that follows closely the definitions of the SQL/MED standard. The close relationship can be attributed to the same roots. DB2's wrapper technology was directly derived from Garlic [JSHL02]. Many of the findings from Garlic had a direct influence on the standard development.

DB2's federated support is separated into two products. DB2 UDB itself defines the part that is necessary to communicate with different wrappers. Additionally, a wrapper for the federated access to other DB2 sources is available via the DRDA wrapper. Wrappers

for other data sources like Oracle Database, Microsoft SQL Server can be used with the WebSphere Information Integrator product[2], which builds on top of DB2 UDB. It also provides an software development kit (SDK) to implement wrappers for data sources that are not directly supported by IBM today. As an example, a wrapper to access Google as a foreign data source was already implemented [Deu04].

The SDK provides the interface to define specialized classes for foreign data wrapper, foreign servers, user mappings, nicknames, and routine mappings. Operations on nicknames do not only include queries; data modifications can also be routed to the foreign data sources. DB2 introduced the concept of type mappings to allow different distinct types to be mapped to the data types at the federated server.

The facilities to manage structured types in general and spatial types in particular are not supported by DB2. The reason is that the query gateway does not allow any transform functions to be used in the communication between the federated server and the foreign data source. Thus, DB2 does not know how to handle the structured data between the database engine and the wrapper. Experimental work to resolve this issue, specifically targeted at spatial data, is presented in [Kac03]. The approach was to modify the DB2 engine directly. When a spatial value is to be fetched from the data source – another DB2 server – the value is prepared in such a way by the wrapper that it uses directly the DB2-internal representation for structured types. That means no transform functions are involved and the solution is only applicable to spatial data originating from another DB2 server. We present a general mechanism applicable to other data sources in Section 7.3. With the wrapper code being available, any data source with spatial data can be connected to DB2 even allowing the push-down of spatial predicates to the data source, i. e. the remote evaluation of such predicates.

7.2.2 Oracle Database

The federated technology implemented in Oracle Database is called *database links*. A database link is comparable to a server registered with the in SQL/MED standardized `CREATE SERVER` statement. Tables and views at the remote server (linked database) can be accessed in the local database not via the notion of registered nicknames but with the slightly deviating syntax `<table>@<database-link>`.

The spatial data types provided by Oracle Spatial [Ora05f] are supported by database links like any other object types that can be created in an Oracle database. However, the requirements for federated support are that the spatial data types at the federated server and the foreign data source carry the same names and, even more important, have exactly the same type definitions. Thus, federated spatial access is only an option for the SDO_Geometry type. Heterogeneous systems cannot be handled because of the quite differing implementations of the spatial types as we laid out in Section 6.1.

[2]The WebSphere Information Integrator was formerly known as DB2 Relational Connect.

An object type has an internal structure and that structure must be considered when the federated server queries the remote table. Internally, Oracle Database invokes a specialized logic that deals with the object types. The single attributes are accessed and transferred separately. The values are then used at the federated server to reconstruct the value of the matching object type.

7.2.3 MySQL

The open source database system MySQL [MyS05] uses so-called *storage engines* that are responsible for the actual data storage. A storage engine acts as a handler for different types of tables. Basic database capabilities like transactional support or different data types are only offered through the various storage engines – or not. For example, spatial data can be managed with the MyISAM storage engine. However, MyISAM does not know the notion of transactions. The InnoDB storage engine would have to be used instead, which does not offer spatial data types. In-memory tables can be maintained with the MEMORY storage engine, and the FEDERATED storage engine is the means to access data in another MySQL system.

The approach taken by the FEDERATED storage engine does not follow the SQL/MED standard at all. Instead, MySQL's storage engine interface is used. Although federated capabilities are available, the already implemented and tested federated functionality is not capable of handling spatial data types and functions even if the foreign data source keeps the data in a MyISAM table.

The open source nature of MySQL would allow the implementation of spatial support directly in the storage engine. Furthermore, it would be possibly to add other storage engines that handle non-MySQL database systems like Oracle Database or SQL Server. However, the main focus in this chapter lies on the facilities specified in the standard. We describe in the subsequent Section 7.3 how federated spatial access can be achieved with implementations closer to the SQL/MED standard than MySQL.

7.3 A Wrapper to Access the GRASS GIS

Accessing federated spatial data does not only involve relational spatial database systems as data sources. The obvious idea is to exploit federated technology to directly access the data maintained in geographic information systems (GISs) and represent its data as relational spatial tables. One such GIS is the Geographic Resources Analysis Support System (GRASS) [GRA05].

The initial work for a GRASS wrapper is described in [Brä05]. Bräu implemented the wrapper for DB2 UDB by exploiting the C++ wrapper SDK. The wrapper extracts the spatial values and its associated non-spatial attributes from GRASS. The spatial data

is encoded in its well-known binary representation (cf. Section 2.3.3) and returned as a binary large object (BLOB). A view is used at the federated server to convert the WKB to the respective ST_Geometry values and, thus, a seamless integration is achieved.

The original GRASS wrapper can perform table scans, projections and selections on the non-spatial attributes. It is not possible to restrict the data based on spatial attributes, i. e. spatial predicates in a query are not pushed down to the data source. The complete spatial processing is done at the federated server. Since that is not sufficient for practical purposes, we extend the GRASS wrapper to support the push-down of spatial predicates.

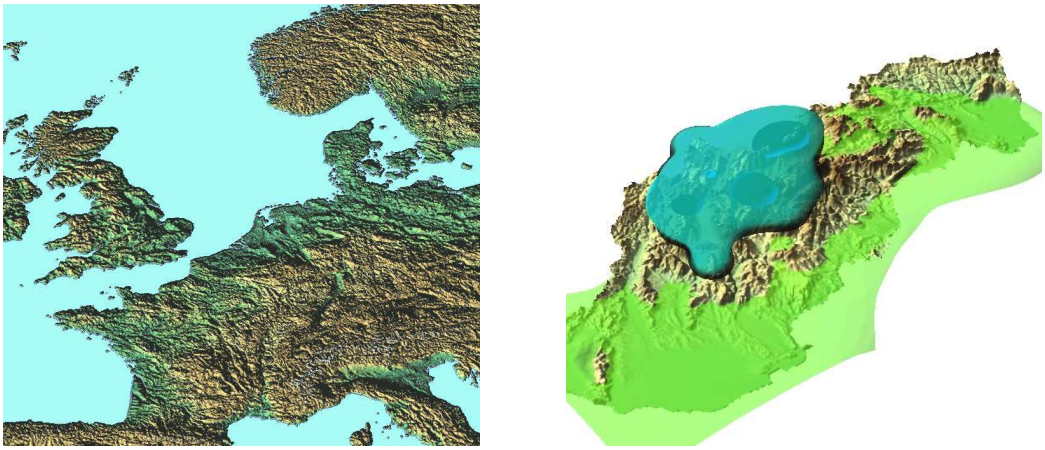
We give an introduction on GRASS in Section 7.3.1. Then the implementation of the GRASS wrapper is described in Section 7.3.2. Some adjustments to the spatial data types in DB2 UDB were necessary to finally facilitate the push-down of spatial predicates. As a result, we made the spatial and non-spatial data in GRASS transparently available as if it were stored at the federated server itself. Naturally, the performance of the GRASS wrapper is not as optimal as if the data were stored directly in the DB2 database, but we show in Section 7.3.3 that it is quite acceptable nevertheless.

7.3.1 Introduction to GRASS

Engineers of the US Army Corps of Engineers' Construction Engineering Research Laboratory (USA/CERL) evaluated the geographic information systems available in the early 1980s. The goal was to select a system that could manage environmental data for the US Department of Defense. It turned out that none of the GIS were up to the task and could satisfy the requirements at that time. A decision was made to develop a new system, which eventually became the Geographic Resources Analysis Support System (GRASS) [NM04]. The ownership of GRASS was transferred in 1995 to the Open GRASS Foundation with Version 4 of the GIS. The foundation merged into the Open Geospatial Consortium since then. The majority of the development is driven by the open source community, but USA/CERL still contributes to GRASS, for example the kernel component for floating point support originates there [Has97].

Today's GRASS Version 6 is comprised of more than 350 different tools and programs. It is possible to manage vector and raster data. Spatial data can be imported from various data sources, constructed or generated, manipulated, visualized or transformed into maps. Two-dimensional as well as three-dimensional information can be handled as *Figure 7.4* illustrates. Figure 7.4(a) shows the overlay of several 2D layers as produced by the tool *nviz*. True 3D visualization is provided by *grass3d* as in Figure 7.4(b), but that is applicable to raster data only.

GRASS provides a common graphical user interface to start the majority of the tools. The tools can also be invoked directly on the command line. Furthermore, the functionality is accessible via a C-API so that it can be embedded in applications. The GRASS wrapper is one example. However, no common API exists for the different tools.



(a) 2D map generated by *nviz* (b) 3D visualization of *grid3d*

Figure 7.4: Visualizing spatial data with GRASS tools

Architecture

Relational database systems had not yet entered the market when the development of GRASS began. The then dominant database systems were hierarchical like IMS/DM and those were not widely used, especially not in the GIS business. Concepts like concurrent and parallel access to the data, a centralized interface, data integrity or the durability of changes on the data, combined with backup and recovery tools were not or only rudimentary existent in GRASS.

The geographic information system does not employ a server process of any kind to synchronize the access to the data it manages. Thus, a client/server architecture cannot be found and other techniques like shared memory are also not adopted. All information, data and meta data, is kept in the file system. Some basic locking is done that way, i. e. the existence of POSIX file locks determines whether a certain set of data (the so-called *gisdbase*) is currently being used. A mapset is a very coarse granularity for locking purposes as is apparent from the mapping of GRASS terminology to relational concepts in Table 7.1. The C-API to read vector information provides parameters to specify different locking modes, but those modes are neither documented nor could any different behavior be observed in experiments.

Summarized, GRASS can mainly be considered as a single-user application. Its architecture as shown in *Figure 7.5* is tailored to that. A typical GRASS installation is comprised of the kernel libraries and many different programs, tools, and modules. All access to the spatial and non-spatial data is routed through the kernel libraries.

There are two basic kernel libraries: the *GRASS library* and the *Database library*. The GRASS library provides some common infrastructure, for example to initialize the environment, and it also establishes the connection to the libraries that take care of the

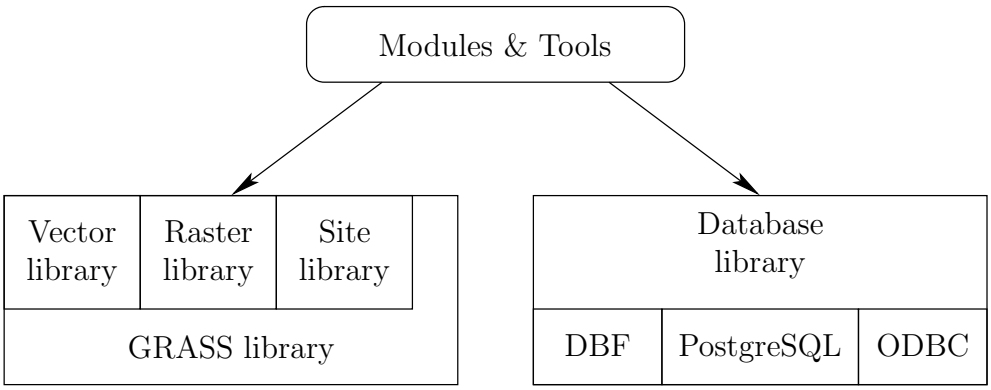


Figure 7.5: Architecture of GRASS kernel libraries

management of the spatial data. The *Vector library* can be used for vector data whereas the *Raster library* handles raster data like satellite imagery. The *Site library* is a GRASS-specialty and dedicated to point data only. Optionally, other libraries for certain image formats can be plugged in to decode raster data like GIF or JPEG [BS95]. Additional storage models like PostGIS [Ref05] can be integrated on that level as well. The combination of all those libraries defines the C-API for the work with the spatial data. Consequentially, the application programming interface (API) is very diverse and the functions of the specific library have to be used – depending on the actual data. In other words, no single and consistent interface is available for applications built on top of GRASS [GRA06, Brä05].

The Database library is the second basic library and it is solely dedicated to the management of non-spatial information. It provides the database management interface (DBMI) that uses a very simplified dialect of SQL to query and manipulate the data. The SQL statements to query the non-spatial data include table scans with projects and simple selection conditions where values in the table can be compared with each other or with constants. Subselects or joins are not allowed. The data itself can be stored in files residing in the file system, in a PostgreSQL database or other systems that come with an ODBC/CLI interface. The access to the various storage systems is mostly encapsulated in DBMI so that a homogeneous interface can be used throughout an application.

Storage Models

Imposed by the GRASS architecture, there are two different ways to physically store spatial data:

1. flat files in the file system, or
2. in PostgreSQL combined with PostGIS.

PostGIS (cf. Section 2.5.6) is an extension for the open source database system PostgreSQL to directly manage spatial data inside this DBMS. Using PostGIS in conjunction with GRASS also allows to store the non-spatial attributes in the same or another PostgreSQL database. At the current stage, this technology is still considered experimental. The file system based storage model exists since the early days of GRASS and it provides a solid and stable foundation. The GRASS wrapper described below adopts this storage model also for its simplicity and the PostGIS storage model is not further discussed.

A directory structure is imposed to organize the spatial and non-spatial data managed by GRASS. *Figure 7.6* illustrates the directory tree. Separate files are created for the vector, raster and attribute data as well as information about the spatial reference system used, along with other information maintained by GRASS and its tools.

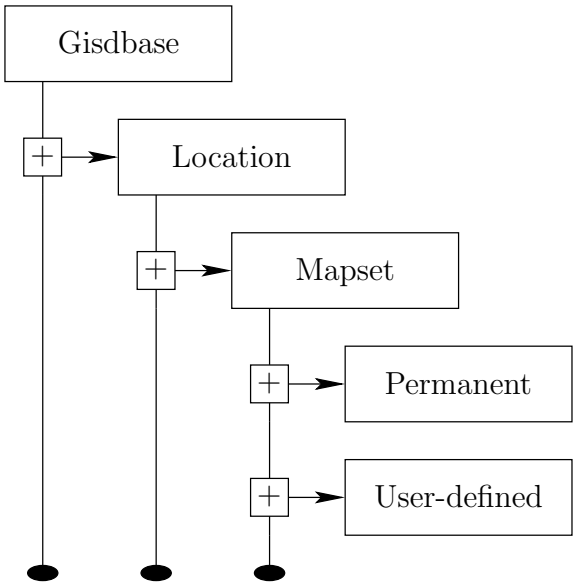


Figure 7.6: Directory structure of the GRASS file system storage model

The main directory is *Gisdbase*. When the GRASS information shall be shared (non-concurrently), the Gisdbase is the granularity that has to be used, for example by exporting that directory as a share of a network file system (NFS). Locking is performed on that level. In terms of relational database systems, the Gisdbase is comparable to a catalog in the sense of [ISO03i]. The subdirectories in the Gisdbase are used for the different *Locations*. A Location uses the same SRS for all its vector or raster data. A mapping of the GRASS terminology to the relational world would equate Locations to databases. One level further down are the *Mapsets* that contain the actual data. Each Location contains at least the Mapset named *PERMANENT*. The PERMANENT Mapset acts as an information schema and it is intended for static spatial data, i. e. data that is not supposed to be changed at all. All other data for the daily business

shall go into user-defined Mapsets. For example, an insurance company could store the street network or county boundaries as reference information in PERMANENT. The address and location of customers will go into another Mapset. Mapsets can be interpreted as schemata in an RDBMS. A set of files and additional internal directories exist within each Mapset, for example to store vectors and raster images. A *Vector* is an array of geometries, the *Features*. Features are points, curves, or surfaces, combined with so-called *Fields*. A field establishes the connection to the non-spatial attributes. A vector can have multiple fields associated with it, implying that each feature in the vector can be linked to non-spatial attributes distributed over several attribute tables, called *Databases* in GRASS.

The mapping of all the GRASS schema elements to relational database systems is summarized in *Table 7.1*. This mapping is essential for the GRASS wrapper as the spatial and non-spatial information stored in GRASS shall be made available like any other table in an RDBMS.

GRASS	RDBMS
Gisdbase	Catalog
Location	Database
Mapset	Schema
Vector	Table
Feature + non-spatial data	Tuple

Table 7.1: Mapping of GRASS terminology to relational concepts

Another particularity not yet explained is the fact that GRASS does not store geometry collections as a single vector element. Instead, the geometry is broken up into its single parts and each part is stored as one vector element. All elements that originally belonged to the same geometry can only be related back by their associated non-spatial attribute data. For example, the ESRI shapefile [ESR98] `country.shp` that is available as sample data for the DB2 Spatial Extender contains the borders of 54 countries in Europe. Importing this shapefile in GRASS will result in a vector that has 3749 elements as each island and each continental area becomes a separate polygon and, thus, a separate vector element. Combined with the multiple fields per vector, that leads effectively to an n-to-m relationship between features and non-spatial attributes as *Figure 7.7* illustrates. To realize this scheme, each feature carries a category identifier (or just *cat* for short). Each part of a geometry collection has the same category identifier. The field identifiers for the vector determine the attribute tables and the cat value selects the rows in those tables.

A dedicated field index provides a fast reverse lookup for the relationship, i. e. to quickly find features for a tuple in an attribute table based on its value in the CAT column. The index is clustered by the field identifiers and each cluster sorts the associations according to the cat values.

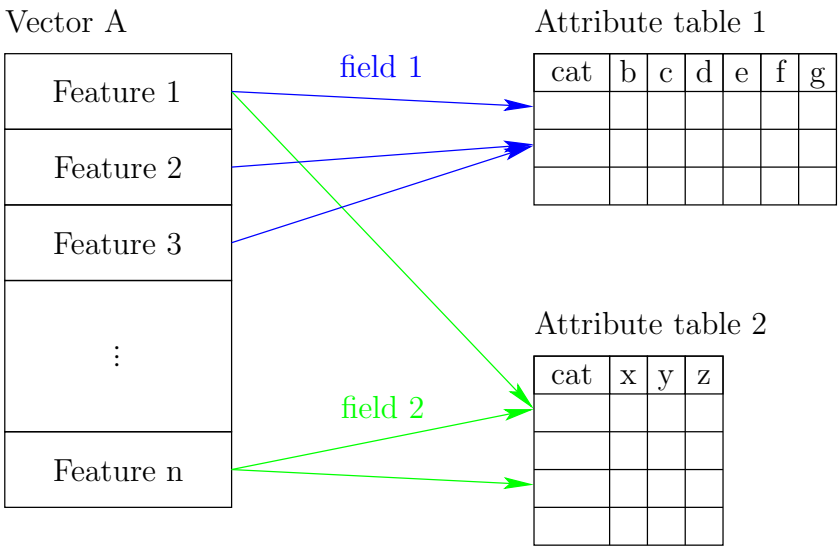


Figure 7.7: Linkage between features and non-spatial attributes

This whole structure must be considered when GRASS data is made accessible through federated technologies in a relational database system. The following section explains how a wrapper for GRASS could be implemented using the facilities defined by the SQL/MED standard, but further restricted to the subset of functionality that is actually provided by DB2 UDB.

7.3.2 Architecture and Implementation of the GRASS Wrapper

This section brings the different pieces together, i. e. DB2's federated infrastructure and wrapper development kit (as we introduced in Section 7.2) is exploited to connect GRASS as a foreign data source to a relational DB2 UDB database. DB2's federated query gateway does not facilitate the handling of structured data as no transform functions can be used. Therefore, we take a different approach for the GRASS wrapper by treating spatial data as BLOBs, which are supported by the query gateway. This technique avoids any changes to the DB2 code base as was necessary in [Kac03]. The details regarding the DB2-internal storage of structured types are not needed either. Thus, a whole new set of opportunities independent of the features in DB2 itself is opened. We apply those features in Section 7.4 to other existing wrappers.

The goals of the GRASS wrapper reach farther than the functionality offered by the initial work done by [Brä05] in that area. A seamless spatial interface shall be established, which adheres to the following conditions:

1. Spatial data at the foreign data source can be queried and is presented as values of type ST_Geometry (or one of its proper subtypes) at the federated server.

2. Data modification at the foreign data source shall be possible for the spatial and non-spatial information.

3. The integration into DB2's access plans should reach a point where not only simple predicates on the non-spatial attributes can be evaluated remotely but also spatial predicates can be pushed down to GRASS.

The first step, the representation of the spatial data in GRASS using the proper spatial data types at the federated server can be achieved by the means of views as *Figure 7.8* shows. The wrapper cannot return values of type ST_Geometry itself, so it resorts to presenting the spatial data as BLOBs. A view over the nickname hides the type mismatch between the BLOBs returned by the wrapper and the spatial data types defined by the DB2 Spatial Extender [IBM04d] by embedding a call to the spatial constructor function that convert the BLOB to its corresponding spatial value.

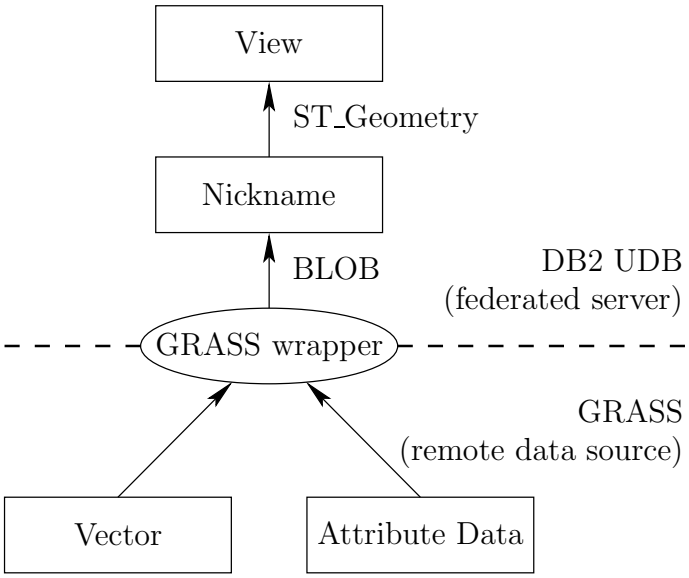


Figure 7.8: Architecture of federated spatial support for GRASS data sources

We describe the specific components of the wrapper subsequently. The query planning requires some additional care for the spatial functionality. That and the logic applicable to handle spatial data at the foreign data source are explained in this context.

Wrapper Class

The wrapper class is only the entry point for the complete logic used for the communication interface between GRASS and DB2 UDB. The class informs DB2 about the properties of the GRASS wrapper, in particular that the code is implemented in

a thread-safe fashion and can be executed concurrently (at least for read-only access). Furthermore, it is not a relational wrapper and it does not use the DB2-specific DRDA protocol for accessing GRASS.

The main purpose of the wrapper class is available via the method *create_server*. The method is used to instantiate the Server class. A server object is used to come up with the execution plan for an SQL statement that accesses federated data via this wrapper.

When a wrapper is registered in a database at the federated server, an instance of the wrapper class is created by DB2. The instance verifies the installation path for GRASS that is given as a wrapper-specific option in the `CREATE WRAPPER` statement as *Listing 7.1* shows. The option is kept in DB2's information schema. When a remote statement is executed at runtime the installation path is needed to properly initialize the environment expected by GRASS. In particular, the environment variables `GRASS_LD_LIBRARY_PATH` and `GISBASE` must be set so that GRASS can find its own shared libraries and programs. The library information in the SQL statement is only needed by DB2 itself as it needs to know which shared library implements the wrapper.

```
CREATE WRAPPER grass_wrapper LIBRARY 'libgrasswrapper.so'
   OPTIONS ( GRASS_INSTALL_PATH '/usr/grass' )
```

Listing 7.1: Registering the GRASS wrapper in the database

Server Class

An instance of the server class is the data structure that represents a single foreign data source, i. e. a GRASS location based on the terminology mapping that we defined in Table 7.1. The server establishes the connection to the respective GRASS location and it creates nickname objects that are used by the federated server to access the GRASS vectors and their associated non-spatial attributes. Another task of a server object is related to the compilation of an SQL statement at the federated server. DB2 consults the server object to come up with the final execution plan.

Server Options The identification of a GRASS location is given to the server when it is registered with the federated database using a `CREATE SERVER` statement as in *Listing 7.2*. The location consists of the path to the GIS database, which is stored in the environment variable `GISDBASE` before GRASS in invoked. Likewise, the environment variable `LOCATION_NAME`, which identifies the location within the GIS database, is also provided as an option to the server.

```
CREATE SERVER grass_server WRAPPER grass_wrapper
    OPTIONS ( GIS_BASE_PATH '/home/grass/data',
              LOCATION_NAME 'sample', SRS_ID '1003' )
```

Listing 7.2: Registering a GRASS Location as foreign data source

Each location contains spatial data in a single SRS. Therefore, the statement to register a server also takes the numeric identifier of the SRS that shall be used for the spatial data retrieved from GRASS and mapped to ST_Geometry values in the DB2 database. Alternatively, the server could employ the SRS mapping techniques presented in Section 6.2. In that case, the server does not need the SRS_ID option. The server will automatically determine the which spatial reference system at the foreign data source matches with which SRS at the federated server. Either way, the Remote Operation class extends the WKB representation retrieved from the data source by appending the numeric SRS identifier applicable at the federated server.

Query Planning Execution plans for SQL statements that involve the access to nicknames referring to data stored at the GRASS server are developed with the direct help of the server object. DB2 employs the *request-reply-compensate* protocol to figure out which fragments of a federated SQL statement can be handled directly by the data source. DB2 identifies portions of the statement that involve the nickname and sends those to the server object as a *request*. The server object evaluates the fragment and returns a *reply* that contains the portions of the fragments (the sub-fragments) that the data source or the wrapper itself can process. The missing logic is then compensated by DB2. A request contains three different types of information:

nickname Each nickname (also called *quantifier*) accessed in the SQL statement is provided to the server. If multiple nicknames occur, it means that a join is to be performed between them. If the foreign data source does not support join operations, the request is rejected by not returning any reply as is done by the GRASS wrapper. DB2 will then break up the fragment into multiple smaller fragments, each accessing just a single nickname. The join is compensated and performed by DB2 itself.

head expression Expressions (columns, functions, or arithmetic expressions) needed at the federated server, for example to return them in the result set of a federated query, are given to the server object. The server returns those expressions in the reply that it can evaluate. At a minimum, the accessed columns of nickname must be part of the reply. Any other expression will be evaluated by DB2 to compensate missing functionality at the data source itself.

predicates The real benefit of federated SQL statements is to evaluate predicates directly at the data source and, thus, to reduce the amount of data processing partaking at the federated server. Each predicate accepted by the server object is added to the reply object. All other predicates are evaluated at the federated server.

The single predicates must be in disjunctive normal form, i. e. linked using the AND operator, in the SQL statement. All other operators cannot be used by DB2 to

separate predicates. Predicates involving said operators must be wholly accepted or rejected by the wrapper. Otherwise, the compensation of predicates not handled by the wrapper would be impaired.

Figure 7.9 shows the different types of constructs in a `SELECT` statement. In addition to the head expressions on the select-list, DB2 will include the columns used in the where-clause in the head expressions. That is necessary if any of the two predicates is rejected by the wrapper and DB2 needs to compensate for it. The compensation is only possible if the expressions used in the predicates are available to the federated server. Therefore, the wrapper has to return the respective head expressions in the reply object. The second predicate in the query illustrates the case where two sub-predicates are combined with the `OR` operator. The GRASS wrapper has to accept or reject both sub-predicates together as this construct appears only as a single predicate.

$$
\begin{array}{rl}
\texttt{SELECT} & \underbrace{column_1}_{\text{Head Expression}} \;,\; \underbrace{column_2}_{\text{Head Expression}} \\
\texttt{FROM} & \underbrace{nickname_1}_{\text{Quantifier}}, \underbrace{nickname_2}_{\text{Quantifier}} \\
\texttt{WHERE} & \underbrace{ST\_Distance(spatial\_column, ST\_Point(-121, 50)) < 100}_{\text{Predicate}} \;\texttt{AND} \\
& \underbrace{(\underbrace{nickname_1.id}_{\text{Head Expression}} = \underbrace{nickname_2.id}_{\text{Head Expression}} \;\texttt{OR}\; column_1 <> column_2 \star 32)}_{\text{Predicate}}
\end{array}
$$

Figure 7.9: Constructs of request objects in an SQL query

Based on the capabilities of the DBMI interface in GRASS, only basic fragments can be handled by the GRASS wrapper. Notably, no joins are supported and the operators in the predicates are restricted to simple comparisons of values of a columns with other columns or with constants. The wrapper will return no reply object for more complicated fragments and the DB2 federated server will break-down the plan into more manageable pieces, for example it will split up joins. An exception to that are spatial predicates, which play a special role in the GRASS wrapper and merit a more detailed explanation.

Spatial Predicate Push-Down DB2's federated infrastructure does not support structured types as the SQL/MED standard specifies. Therefore, the spatial data from GRASS cannot be returned directly as values of type ST_Geometry. Instead, the GRASS wrapper extracts the spatial data from the GRASS vector and converts it to its WKB

representation. A view V over the nickname N transforms the BLOB data to a spatial value, providing completely transparent access to the GRASS data. As a result, the nickname is not directly used by applications. The spatial predicates used in SQL statements accessing the view are the ones that need to be pushed down.

The constructor function named *ST_Geometry* is called in the definition of V for the conversion from WKB to the actual geometry. A two-step implementation applies for the constructor. First, a routine coded in SQL is used to query the spatial information schema, namely the view ST_SPATIAL_REFERENCE_SYSTEMS, to get all the parameters of the SRS that shall be associated with the final geometry. The second step is the invocation of an external routine that takes the WKB along with the SRS and builds the internal representation. A transform function comes into play to actually create the geometry value. When V is accessed, its definition is expanded into the SQL statement. Additionally, the first function in the construction process is expanded as well. The expansion for the *ST_Intersects* method yields effectively the SQL fragment in *Listing 7.3*. The *query_window* in the fragment represents a given spatial value with which the spatial column WKB_COLUMN from the GRASS data source is tested for intersection.

```
ST_Intersects(query_window, ST_Geometry(wkb_column,
      ( SELECT definition, ...
        FROM   st_spatial_reference_systems
        WHERE  srs_id = 1003 )))
```

Listing 7.3: Expansion of constructor function in call to *ST_Intersects*

A push-down of spatial predicates like *ST_Intersects* is not possible with this implementation. An access to a local table is required in the nested function call and the DB2 query gateway is not aware that the table ST_SPATIAL_REFERENCE_SYSTEMS may also be available at the foreign data source and could be accessed there. Thus, the request passed to the wrapper server object does not even include the predicate with the spatial function as DB2 already decided beforehand to evaluate it at the federated server. This issue can be circumvented with an adjustment to the definition of the spatial constructor function *ST_Geometry*. Instead of fetching the information about the associated SRS from the spatial information schema, the function gets only the numeric identifier of the SRS as input parameter. Then the required information is accessed from inside the function via an SQL statement, requiring that the function is declared as `READS SQL DATA`. This difference might appear negligible, but it is sufficient that the DB2 query gateway does not detect any access to a local table for the input parameters of the *ST_Geometry* function.

The second restriction for the spatial predicate push-down stems from the transform functions that are employed by the DB2 Spatial Extender to transfer spatial values between the DB2 engine and the extender code itself. Once a routine involves a transform function, the query gateway will not consider it for push-down. That is due to the fact that transform functions are compiled into the statement. Regardless of the actual

transformation, DB2 will use a temporary table during construction to collect all the values that are to be populated into the type attributes. This temporary table is considered as a local table and – exactly as with the spatial information schema – the access to it interferes with the push-down of the spatial function. Getting rid of the transform functions is the only solution to work around this issue.

No local base table, view or temporary table can be used. That also rules out table functions. The results of table functions are collected by DB2 in temporary tables for further processing. Otherwise, an external table function could receive the WKB representation from the GRASS wrapper as input and return all attributes in a table with just a single row. Another function wrapped around the table function would create the spatial value and populate the attributes based on the values in the single row.

That leaves us only two options. Either the wrapper provides the attribute values as columns in the nickname and the view projects those columns away, or a set of functions is employed where each function gets an extended WKB as input. The WKB includes the numeric SRS identifier at the end so that it is self-contained. Each function computes a single attribute. Both approaches are identical from a functionality point of view when a spatial predicate is pushed down. If separate functions are used for the 14 attributes (cf. Section 6.1.1), those functions are nested in the spatial predicate and they all have to be pushed down to avoid any local evaluation at the federated server. Unfortunately, the external functions are only considered for predicates and not in head expressions. The DB2 query gateway does not include the function calls there and, thus, prevents the GRASS wrapper to produce those values itself. Despite the performance penalty for the function invocations, the function-based approach is adopted subsequently to avoid changes to the relational schema of the nickname pointing to the GRASS data.

Listing 7.4 shows the fully expanded SQL fragment for the construction of a linestring. If this fragment is nested in a call to *ST_Intersects* or other spatial comparisons, it is contained in a predicate in the request object. This logic for the construction of a spatial value at the federated server can be nested in a function written in SQL to simplify its interface. The function will be automatically expanded by DB2 when invoked.

```
ST_LineString()..srid(GetSrsId(srsId))..
   numPoints(GetNumPoints(wkb_column))..
   geometry_type(GetGeomType(wkb_column))..
   xMin(GetMinX(wkb_column))..xMax(GetMaxX(wkb_column))..
   yMin(GetMinY(wkb_column))..yMax(GetMaxY(wkb_column))..
   zMin(GetMinZ(wkb_column))..zMax(GetMaxZ(wkb_column))..
   mMin(GetMinM(wkb_column))..mMax(GetMaxM(wkb_column))..
   area(GetArea(wkb_column))..
   length(GetLength(wkb_column))..
   points(GetPoints(wkb_column))
```

Listing 7.4: Constructing a linestring without transform functions

The construction of spatial values is not the only modification required in the DB2 Spatial Extender. The comparison functions to be pushed down to the GRASS data source must adhere to the same conditions as the constructor, i. e. no transform functions must be involved and no tables stored locally at the federated server can be accessed. *ST_Intersects* and related routines have to be reimplemented to avoid the transform functions. That can be achieved by two nested functions. The first accesses all necessary attributes of a geometry and passes those values as input parameters to the second, which performs the actual calculation. In other words, the logic of the transform function is made explicit.

Applying the described mechanisms allows finally the push-down of spatial predicates. The SQL statement shown in *Listing 7.5* is given to the federated server. It queries the view V and compares the geometries at the foreign data source with another geometry provided by the client application. The above constructor logic is gathered in the *Construct* function. The BLOB used in the query is the extended WKB representation of a point. It is fully self-describing, i. e. it includes the byte order, type identifier and the coordinates of the point $(10, 10)$ (set in blue), and beyond that it also includes the numeric SRS identifier (set in green) for the resulting geometry in the last four bytes. The column SPATIAL_COLUMN represents the spatial values in GRASS.

```
SELECT id
FROM   v
WHERE  1 = ST_Intersects(spatial_column, Construct(BLOB(
   x'0101000000000000000000244000000000000002440EB030000')))
```

Listing 7.5: Query against view V over nickname N

The query results in the fragment shown in *Figure 7.10* that is sent as request from DB2 to the GRASS server object. The operator *ST_INTERSECTS* requires four input parameters, two for each geometry: the points attribute, which contains the internal encoding of the geometry definition, and the numeric SRS identifier. The figure shows the subtree for only one of the observers accessing those attributes as the others have a very similar structure. The operators set in yellow refer directly to the functions that are used to extract the respective attribute values from the WKB. There are 15 such operators as 15 functions are invoked – one for each attribute. Only four of them are shown to keep the figure comprehensible. The blue operator represent the constant expression with which the result of the spatial function *ST_Intersects* is compared for equality. The chain of *MUTATOR* operators starting with the *CONSTRUCT_ADT* operator is intended to create the structured value and populate its attributes. The *UNWRAP* operators access the DB2-internal structure that represents the geometry. The operator *SPATIAL_DATA* refers to the spatial column of the nickname, through which the extended WKB is available.

The GRASS server object analyzes the predicate to verify if it adheres to the expected structure shown in Figure 7.10 and accepts it if it fits. The predicate is added to the reply object and the federated server will not evaluate it. The server object will then

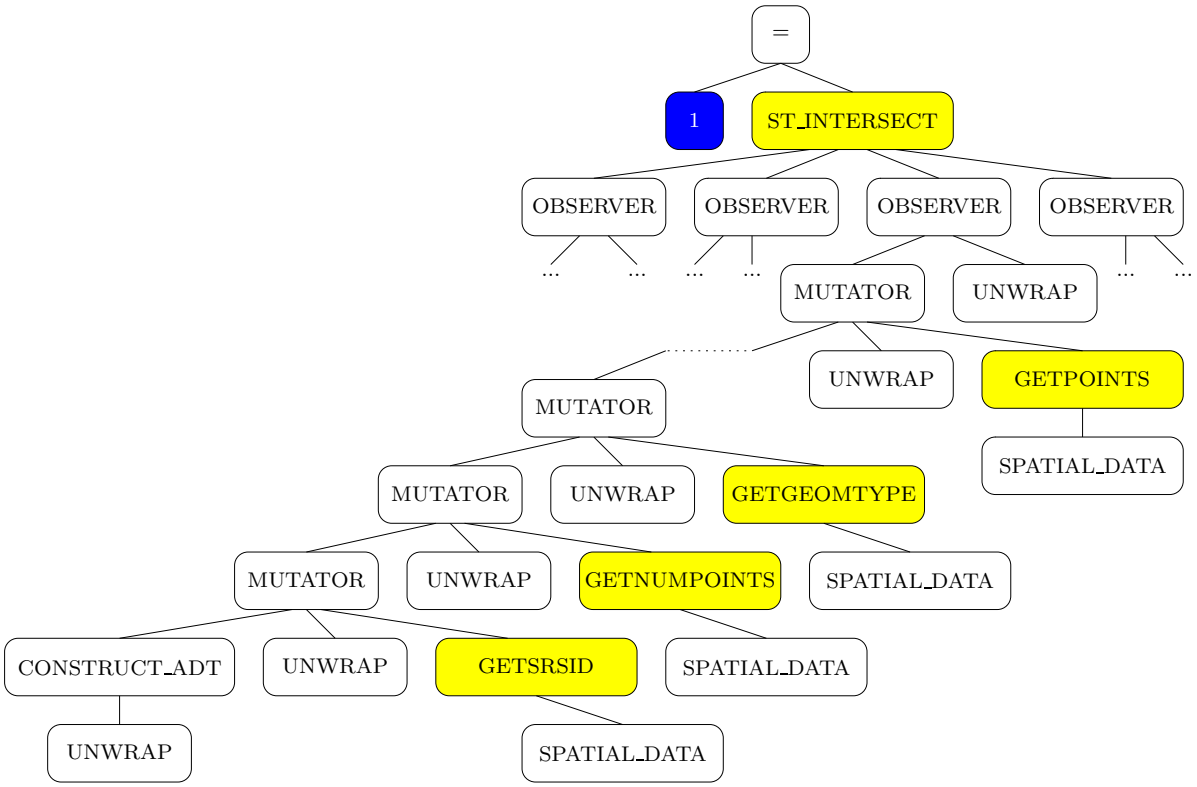


Figure 7.10: Predicates in request object provided to server object

store the information about the query geometry and the spatial column, i. e. the vector in the GRASS system together with the actual operator in an execution descriptor. The execution descriptor is a means to pass wrapper-specific information from the planning to the execution phase. The GRASS vector and the relevant non-spatial columns are encoded in the descriptor as well.

The result from the statement compilation is an access plan that is processed jointly by DB2's runtime component and the wrapper. The plan for the query in Listing 7.5 is illustrated in *Figure 7.11*. Most notably, there is no filter step between the remote push-down (RPD) and the final return of the result. That proves that the spatial predicate is indeed not evaluated by DB2. An instance of the Remote Operation class of the GRASS wrapper must take care of the predicate evaluation at the foreign data source. To that end, DB2 will pass it the execution descriptor that was built during the planning phase.

Nickname Class

The purpose of a nickname is to represent a GRASS vector and its associated non-spatial attributes in a relational fashion to the federated server. Once a nickname is registered

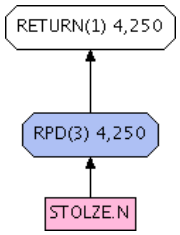


Figure 7.11: Access plan for a federated query

in the federated database, it is conceptually not different from any other base table or view. It consists of a set of columns and each column has a declared data type.

The nickname class in the GRASS wrapper is merely a descriptor for the remotely stored data. The class implements methods that allow specific options to be associated with the nickname, namely the two options that identify the mapset and the vector can be set or modified as *Listing 7.6* demonstrates. The third option identifies the spatial column by its name as relying on the data type is not possible. Other columns could be of type BLOB and those should not be interpreted as spatial columns. The column information like data type and nullability can either be provided in the `CREATE NICKNAME` statement or the GRASS wrapper will establish a connection to GRASS and gather the desired information from there directly. The first approach allows for specialized type mappings like mapping a string in GRASS to the CHARACTER data type. The second variation is preferred for its much simpler user interface.

```
CREATE NICKNAME grass_table FOR SERVER grass_server
   OPTIONS ( MAPSET_NAME 'usa', VECTOR_NAME 'cities',
             SPATIAL_COLUMN 'spatial_data' )
```

Listing 7.6: Registering a nickname pointing to a GRASS vector

Connection Class

The connection class is a helper construct to establish a communication channel between the wrapper and the foreign data source. A connection is initiated by the GRASS server object whenever the DB2 federated server wants to access the data source.

The class is responsible for the connection management and for that it provides methods to open or close a connection. If GRASS had been able to support transactions, the information about transaction boundaries would also be sent by the connection class to the foreign data source. That would happen if the SQL statements `COMMIT` or `ROLLBACK` are processed, or if the client connected to the federated server abends or disconnects and any currently running transactions are rolled back implicitly. However, GRASS has no notion of transactions, so the respective methods are not implemented in the GRASS wrapper.

User Mapping Class

The federated query gateway of DB2 follows the specifications of the SQL/MED standard and allows the definition of user mappings. The user mapping declares the credentials that shall be used when connecting to the foreign data source for which the mapping is created. The GRASS wrapper uses only a default implementation for the user mapping class as GRASS does not come with any mechanisms to distinguish users.

Remote Operation Class

A very important part of the GRASS wrapper is the actual execution of a remote operation. An instance of the remote operation class is responsible for that. The logic of the class is very closely tied to the server class as it has to execute the operation that was planned by the server.

A remote operation can either be a query (`SELECT` statement) or a data modification (`INSERT`, `UPDATE`, or `DELETE`) statement. The remainder of this section is concerned with queries. Data modifications can be handled by converting the spatial data to a BLOB and an instead-of trigger can provide this conversion. *Figure 7.12* illustrates this configuration. If a data modification statement includes spatial predicates, the same considerations as for queries apply.

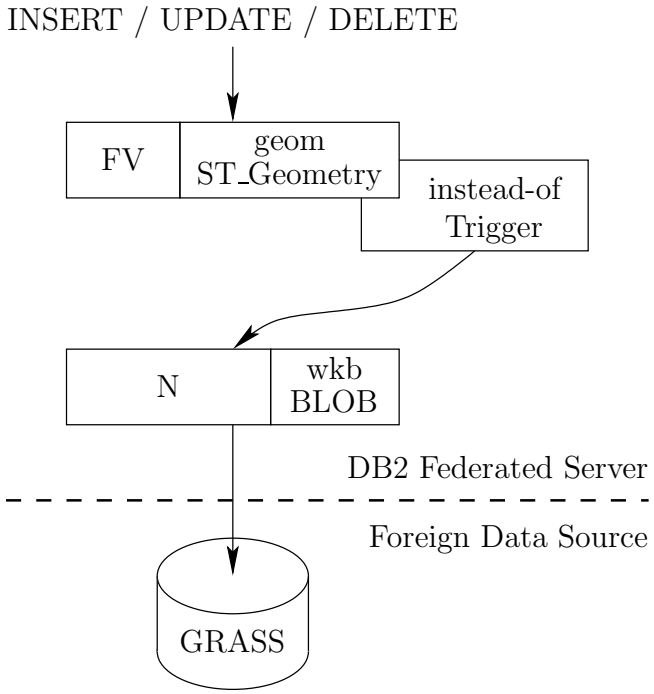


Figure 7.12: Handling spatial data in federated DML statements

The remote operation class implements a cursor-like interface for queries. When the class is instantiated, it receives a connection object and a so-called runtime-data object as input parameter. The connection object, created by the GRASS server before, is used to open a channel to GRASS. The runtime-data object contains the full information about the fragment of the original SQL statement that the wrapper has to execute, including the execution descriptor built by the server object during the planning phase.

For remote queries, all necessary preparations are done before the remote operation is started by opening a remote cursor. DB2 will subsequently fetch the single tuples from the wrapper. The GRASS wrapper retrieves the data from GRASS and transforms it according to the expectations of DB2's query gateway component. All non-spatial attributes are simply passed through. The spatial data in the GRASS-specific format is parsed and converted to the extended WKB representation and given to DB2 as a BLOB. When all data for the query is returned, DB2 informs the remote operation object to close the cursor at the appropriate point in time. That happens when the cursor opened over the result set at the federated server is closed explicitly or at the end of the transaction.

Due to the situation that GRASS stores the spatial and non-spatial data not together, two separate requests have to be issued by the wrapper. One request fetches a tuple of non-spatial attributes and the second request retrieves the corresponding geometry. The geometry may consist of several pieces because GRASS stores the pieces separately as Figure 7.7 illustrated. The single pieces must to be merged together when the extended WKB is created. Therefore, it is always necessary to scan over the non-spatial attributes as that is the only means to detect which elements of a feature belong together. The GRASS wrapper implements a sort/merge join. The features and also the non-spatial attributes are ordered based on the linkage value that connects both, i. e. the *category*. Any predicates that restrict the respective set are applied. A minimum bounding rectangle can be used to filter the features indexed in an R-Tree [Gut84]. The non-spatial attribute data is retrieved via the DBMI interface [GRA06] and any conditions as well as the ordering criteria are included in the (simplified) SQL statement processed by DBMI. A sweep with ascending category values is performed during the subsequent fetch operations to find the matching pairs of features and non-spatial attributes.

7.3.3 Performance Results

The GRASS wrapper has been proven to provide the desired functionality. The spatial data stored in and managed by GRASS can be treated like any spatial data directly present in a DB2 database. A major practical issue is always that the functionality can be implemented with an acceptable performance. This section presents the results of the performance measurements based on the described GRASS wrapper. The tests were run on a ThinkPad T30 with a 2 GHz Intel processor and 1 GB of physical main memory. GRASS in Version 6.0.2RC4 was installed on the same machine as DB2 UDB Version 8.2.2.

It is intuitively understood that the GRASS wrapper cannot seriously compete with the spatial data being stored in the DB2 database using the DB2 Spatial Extender. The federated infrastructure introduces a certain overhead and the performance of GRASS is another deciding factor.

We used the road network of several states of the United States of America as sample data. The spatial data and their associated non-spatial attributes are available in shapefiles from the Bureau of Transportation [Bur04]. The six US states Connecticut, Delaware, New Mexico, Michigan, Pennsylvania, and Wyoming were chosen together with the District of Columbia as representatives with the number of roads ranging from a mere 14,762 (D.C.) to 842,099 (Pennsylvania). Other states like California or Texas have even more roads, i. e. 1,589,712 in California and 2,259,314 in Texas. Unfortunately, their data could not be included in the tests because GRASS is not able to import the shapefiles. GRASS tries to load the complete data of those shapefile into main memory before the imported data is written to its own storage paths. Given that the Californian roads already occupy 190 MB for the spatial data and 234 MB disk space for the non-spatial attributes, the system quickly starts thrashing and the whole virtual memory of 2 GB available on the test system was not sufficient to contain those data and the GRASS-internal structures being built.

The first test established a base line for behavior of the federated system. *Figure 7.13* illustrates the runtimes for the retrieval of the complete set of roads for the various states. SQL statements fetch the data like the one shown in *Listing 7.7* does for the state Wyoming. The invocation of the spatial method *ST_NumPoints* is to ensure the retrieval of the spatial information as DB2 may otherwise optimize it away. We access a view and the extended WKB representation is converted to the respective ST_Geometry values the way we already explained.

```
SELECT geometry..ST_NumPoints(), name
FROM   wy_view
```

Listing 7.7: Query to access spatial data in GRASS

Two different implementations to join the spatial and non-spatial data in the GRASS wrapper are compared, namely the sort/merge join described before and the nested loop implemented by Bräu [Brä05]. Additionally, the access to spatial data stored in a base table in the DB2 database is included as reference point.

The nested loop was only measured for the states with a low number of roads because the performance of this approach comes with a quadratic runtime. The reason for its bad performance lies in the lack of index support for the non-spatial attributes in GRASS. Bräu [Brä05] retrieves a single geometry from GRASS and then executes an SQL query against DBMI to fetch the associated information for just that geometry. The SQL query always amounts to a full table scan in DBMI. The more tuples are to be returned by the wrapper, the more table scans are needed. And the more data is managed by GRASS,

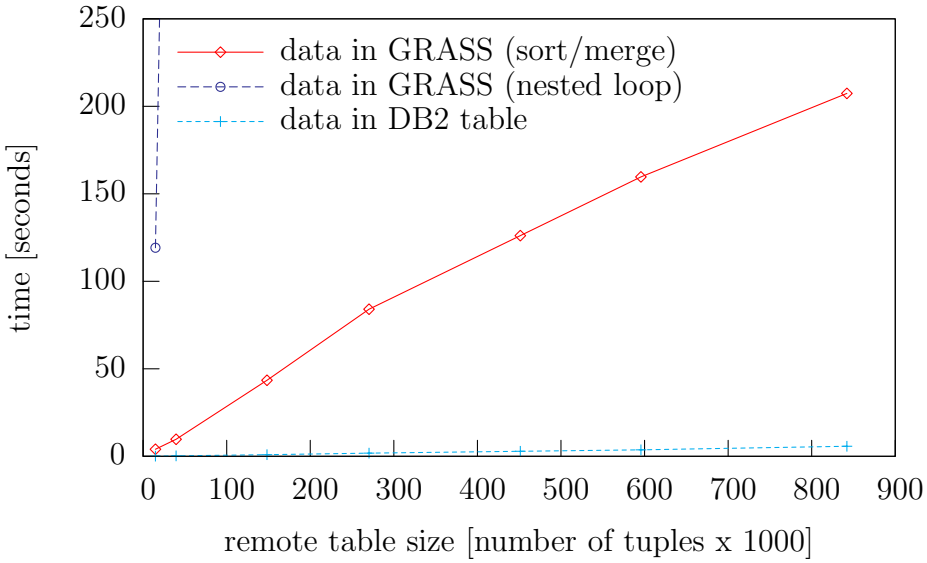


Figure 7.13: Time to access spatial data

the slower each of those table scans will be. The sort/merge avoids this issue by only performing a single scan over the GRASS vector and a single scan over the non-spatial attributes. The overhead for sorting the results of both scans is negligible.

The push-down of spatial predicates is a major feature in the GRASS wrapper. It also has a substantial impact on the performance as Figure 7.14 demonstrates. The queries run against the federated server use the *EnvelopesIntersect* function that tests the minimum bounding rectangles (MBRs) of the geometries for intersection only. Thus the function is semantically equivalent to *ST_Intersects* if only axis-parallel rectangles are considered. The function is provided as part of the DB2 Spatial Extender and its occurrence in a federated query is detected by the GRASS wrapper as was explained before in Figure 7.10. The wrapper uses the GRASS API to filter the elements of a vector. An R-Tree takes care of the spatial indexing. Each of the queries used for the measurements returns about 3% of the roads in the respective state. *Listing 7.8* shows the query for Wyoming that results in 8052 of the 270334 roads being selected.

```
SELECT geometry..ST_NumPoints(), name
FROM   wy_view
WHERE  db2gse.EnvelopesIntersect(geometry,
          -108.00, -107.00, +43.00, +44.03, 1003) = 1
```

Listing 7.8: Query with spatial predicate push-down to GRASS

Figure 7.14 compares the evaluation of the spatial predicate in GRASS with its evaluation at the federated server. In the latter case, the execution time is dominated by the retrieval of the data from GRASS and the predicate evaluation itself does not have

a significant impact. The push-down of this particular spatial predicate reduces the execution time by 66% because much less data is retrieved. Of course, the performance of spatial data stored in the DB2 database cannot be matched.

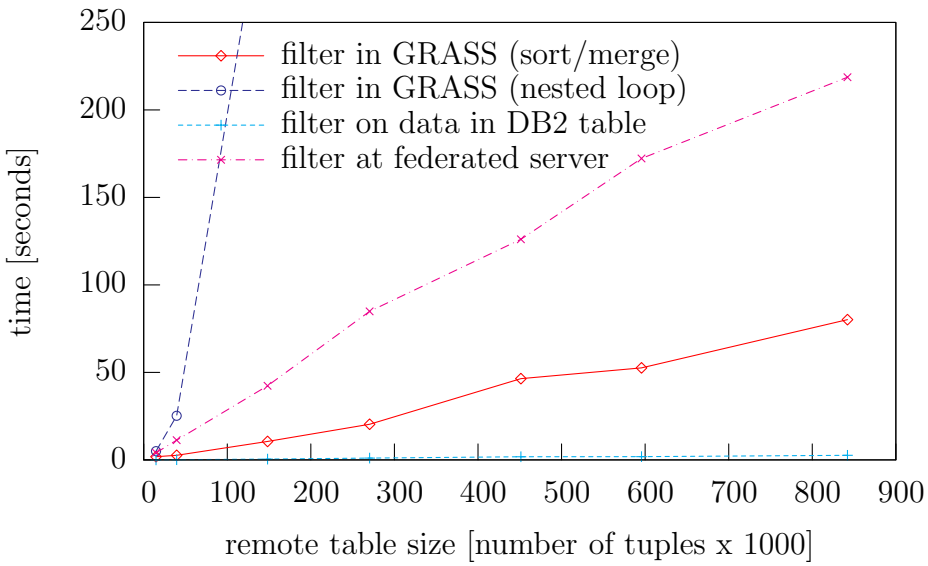


Figure 7.14: Evaluation of spatial predicates

Summarized, we have shown the practical merits of the GRASS wrapper. In particular, the push-down of spatial predicates is very beneficial. However, GRASS itself imposes several obstacles, beginning with the fact that it does not scale and not ending at the storage model that physically separates spatial and non-spatial data without index-based access to the latter.

7.4 Applying the Techniques to other Wrappers

The GRASS wrapper demonstrated that the access to spatial data is feasible. However, spatial data can be stored in a wide variety of systems, including spatial relational databases. Therefore, we carry the techniques developed for the GRASS wrapper to other, existing wrappers that handle spatial data from other foreign data sources. Our goal is to accomplish the access without changing or reimplementing the respective wrapper. By means of an example, the federated access to spatial data stored in a foreign DB2 data source will serve as a reference.

The way to access spatial data in a foreign DB2 database follows closely the approach taken by the GRASS wrapper, i. e. the spatial data of the remote system is represented as BLOBs at the federated server. Two options are available for that purpose. A view over

the remote table can be used to convert the spatial data to a BLOB and the nickname at the federated server refers to the view only. We explain this setup in Section 7.4.2. The second idea, presented in Section 7.4.3, is to implicitly use the transform group *DB2_PROGRAM* for the conversion to a BLOB. Common to both approaches is the handling of the federated spatial data at the federated server. Given our restriction of using the existing wrappers unchanged, the technology from the GRASS wrapper needs to be adjusted at the federated server and we explain in Section 7.4.1 what is actually needed and which effects it has on applications that deal with federated spatial databases.

7.4.1 Setup at the Federated Server

The server class of the GRASS wrapper is tailored to handle SQL statements at the federated server that access the view over the nickname. In particular, the wrapper detects and handles predicates like the one in Figure 7.10 that involves a complex expression to build a spatial value based on the column of the nickname with the (extended) WKB representation. Such expressions are not supported by the various wrappers shipped with the IBM WebSphere Information Integrator, especially not the DB2 (DRDA) wrapper.

In order to ensure that a wrapper does not accept a predicate that it cannot push-down to the foreign data source, the wrapper analyzes the complete tree structure of the predicate. As part of that, the wrapper will detect the `CONSTRUCT_ADT` and `MUTATOR` operators. Those operators are not known to the wrapper, so the complete predicate is rejected and the federated server will have to compensate it, i. e. execute it locally.

As a result of the analyzation and rejection there is no way to use the spatial predicates against the spatial column in the view at the federated server as the constructor in the view definition would always come into play. A solution to avoid the constructor of the structured type is to not apply the spatial predicate on the spatial column. Instead, an additional column is added to the view and that column carries through the extended WKB representation from the nickname as it is. Thus, the view FV at the federated server has to be defined as in *Listing 7.9*.

```
CREATE VIEW v AS
  SELECT Construct(wkb) AS geom, name, wkb
  FROM   n
```

Listing 7.9: View definition at the federated server

The spatial functions that shall be pushed down to the foreign data source have to be applied to the WKB column. For that, those functions must be overloaded to accept values of type BLOB as input parameter. For example, an overloaded variation of the function *EnvelopesIntersect* with the first parameter of type BLOB, the next four parameters of type DOUBLE, and the last parameter of type INTEGER can be declared. It is mapped to the function with the same name at the foreign data source. There, the first

parameter is of type ST_Geometry while the other parameters have the same data type. Routine mappings inform the wrapper that the BLOB-related routines are mapped to routines with the same name at the data source. The wrapper is not aware about the different data type for the routine parameters, however. The function resolution will be solely taken care of by the foreign database system.

Figure 7.15 summarizes the setup at the federated server. It is apparent in the figure that a client application needs to be aware of this situation to invoke the spatial functions on the WKB column to get spatial predicates pushed down and to filter the rows of the result set already at the foreign data source. However, the selection of spatial data from the view at the federated server has to be directed to the GEOMETRY column. Thus, a seamless integration cannot be achieved.

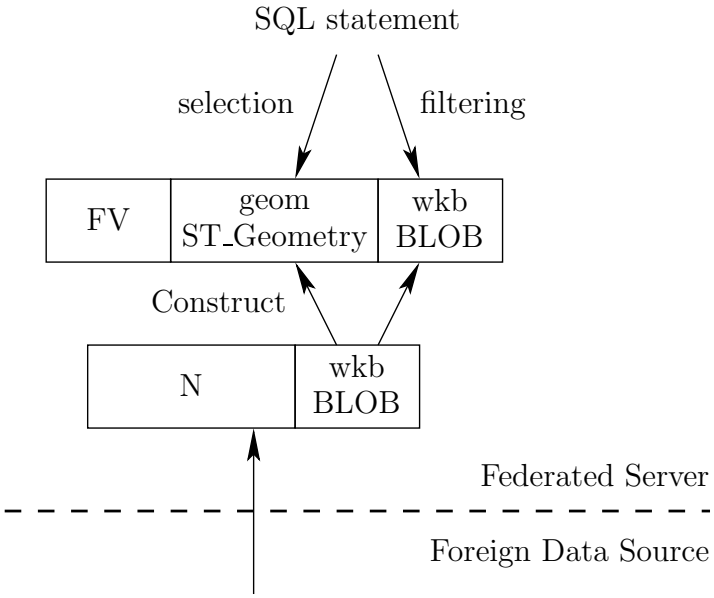


Figure 7.15: Setup at the federated server

The approach to use two distinct columns of different types is only due to the lack of support in the respective wrappers. If the wrapper implementation accepts expressions that construct the spatial values at the federated server, the additional column with the extended WKB could be omitted in the view FV, exactly as the GRASS wrapper already demonstrated.

7.4.2 Views at Foreign Data Sources to Mask Spatial Types

The technique to hide the involvement is spatial data types in the federated setup by introducing views at the foreign data source and at the federated server is very similar to the BLOB-based spatial replication that we describe in Section 8.4.1. A view is created

over the remote table in such a way that the view definition takes care of the conversion of the spatial data to its WKB representation. A nickname referring to this view is declared at the federated server. Thus, the nickname is not aware of the underlying spatial data type and it only deals with BLOBs in the traditional and well-established manner. The final step is to create a view over the nickname to convert the WKB back to the respective ST_Geometry values. *Figure 7.16* illustrates this setup. The view V converts the spatial data in the column GEOM to its WKB using the method *ST_AsBinary*. The nickname N accesses the remote view and the view FV at the federated server converts the BLOB data in the column WKB back to the ST_Geometry values.

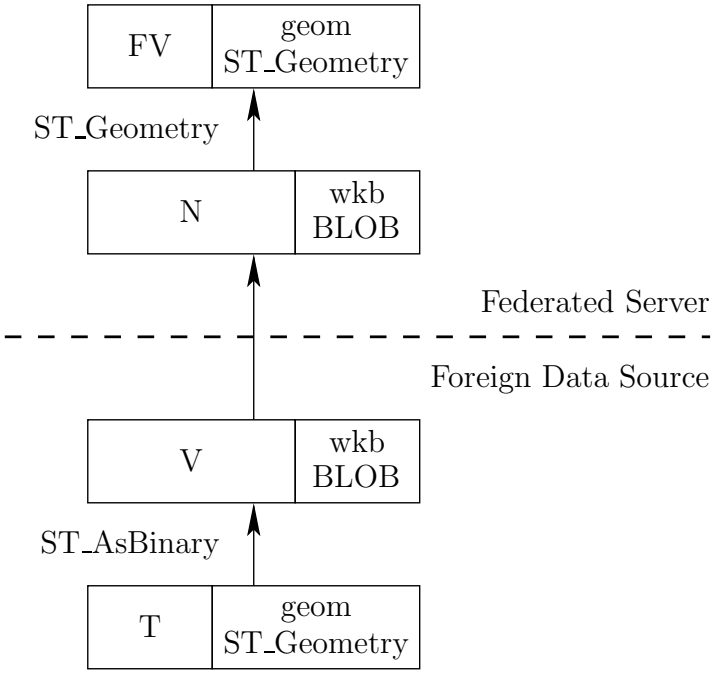


Figure 7.16: Employing views for federated spatial access

This technique can be used with any user-defined type, regardless of the foreign data source involved. The only requirement imposed on the data source is that it implements the concept of views, which is usually a given for relational database systems that also provide support for spatial functionality.

A major draw-back of the view-based technique is that the push-down of spatial predicates is not possible. The method *ST_AsBinary* is contained in the definition of V to convert the spatial values to their WKB as *Listing 7.10* demonstrates.

```
CREATE VIEW v AS
  SELECT geom..ST_AsBinary() AS wkb, name
  FROM   t
```

Listing 7.10: View definition to mask spatial data types

Any predicate that is pushed down from the federated server to the data source has to operate on the results of the *ST_AsBinary* method. Thus, the two queries on T and V in *Listing 7.11* are identical from the perspective of the remote database system, given that the view definition is compiled into the query. Both queries use an overloaded version of the DB2 Spatial Extender function *EnvelopesIntersect* to filter the geometries based on their spatial properties (MBB). The variation used here operates on BLOBs with the WKB representation and not ST_Geometry values themselves.

```
SELECT wkb, name
FROM   v
WHERE  db2gse.EnvelopesIntersect(wkb,
          -108.00, -107.00, +43.00, 44.03, 1003) = 1

SELECT geom..ST_AsBinary() AS wkb, name
FROM   t
WHERE  db2gse.EnvelopesIntersect(geom..ST_AsBinary(),
          -108.00, -107.00, +43.00, 44.03, 1003) = 1
```

Listing 7.11: Queries at remote data source

The first parameter of the *EnvelopesIntersect* function does not identify a column in table T. Unless the database manager knows how to unroll the call to the method *ST_AsBinary* and to map the overloaded *EnvelopesIntersect* function to the function with the same name that takes geometries as input, the system cannot exploit any existing spatial indexes like the DB2 Spatial Extender's grid index [IBM04d] or the Informix R-Tree index [IFX04b] on the column GEOM in T.

The evaluation of predicates on the WKB as in Listing 7.11 may still be beneficial because the rows not satisfying the predicate do not have to be sent from the data source to the federated server. However, we do not pursue this path any further because the alternative relying on default transform groups allows the push-down of spatial predicates, even though such defaults are specific for each database system.

7.4.3 Employing the Default Transform Group

The DB2 Spatial Extender implements the three transform groups that are specified in the SQL/MM spatial standard (cf. Section 7.1.3). Additionally, two transform groups are available:

- *ST_Shape*, and
- *DB2_PROGRAM*.

The transform group *ST_Shape* is similar to the *ST_WellKnownBinary* group in that it implicitly converts a geometry to a BLOB. The difference can be found in the actual encoding of the geometry, which is the shape format and not in the well-known binary representation.

The *DB2_PROGRAM* transform group is special from the perspective of the DB2 engine. If a query involves a spatial column without applying any routine on the values in that column (and if no other transform group was explicitly specified), then DB2 falls back to the *DB2_PROGRAM* transform group to determine how to convert the data before it is sent to the application. The DB2 Spatial Extender implements functions for this group to always provide implicit transform support. That can be taken advantage of in federated spatial environments. During the execution phase of a federated spatial query, the wrapper will directly fetch the data from the foreign source table and the transform group converts implicitly the spatial data to a value of type BLOB. The BLOB can then be accessed at the federated server like any other BLOB without the wrapper knowing the true origin.

An issue arises with the encoding of the geometry data in the BLOB produced by the transform group. It is an internal encoding and it is not documented. The transform group identifies the function *AsSdeTrans* as the means to convert a geometry from its SQL representation to the external format. In turn, that function invokes the method *AsSde* declared on the ST_Geometry type. The counterpart for this method is the *GseGeomFromSDE* constructor function as can be derived from the DB2 information schema. Tests have shown that BLOBs generated with *AsSde* can be used as input for this constructor to generate the very same geometry again. With this knowledge, the internal encoding can be used as the vehicle to communicate spatial data between a foreign DB2 data source and a DB2 federated server as *Figure 7.17* depicts.

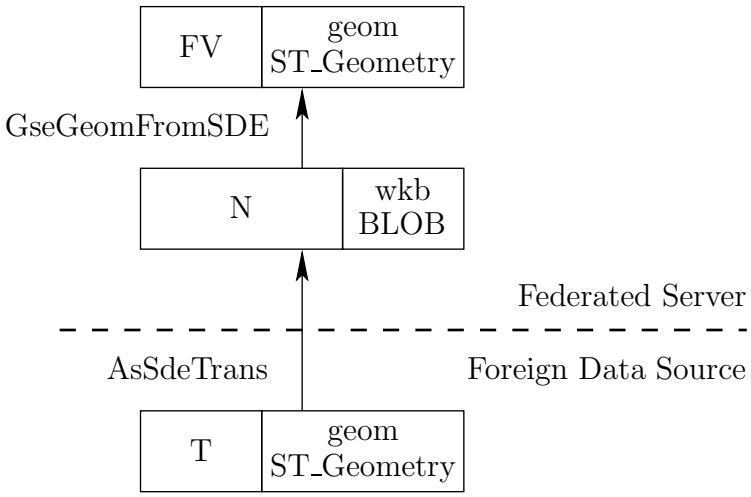


Figure 7.17: Using the implicit *DB2_PROGRAM* transform group

This setup cannot simply be created. When the SQL statement `CREATE NICKNAME` is executed at the federated server, the DB2 wrapper retrieves the information about the declared data types of the columns in the remote table. The wrapper will find the spatial data type and reject the registration of the nickname as it does not know how to deal with this type. The wrapper does not consider transform functions at all and it does not know that querying the spatial column effectively yields values of type BLOB. Therefore, a work-around is necessary during the registration phase. A dummy table that has the same schema as the table T – except for the data type of the spatial column – is used as remote table to which the nickname refers. Instead of the spatial column, a column of type BLOB is used. This column carries the same name as the spatial column. Thus, the wrapper sees only supported data types during the registration of the nickname and will store those information in the information schema at the federated server. After the `CREATE NICKNAME` statement is completed, the dummy table is replaced with the actual spatial table without the federated server noticing the difference because the transform group *DB2_PROGRAM* covers the handling of the spatial types at runtime.

The major difference to the setup in Figure 7.16 is that the wrapper at the federated server directly accesses the remote source table without any additional explicit function calls being involved in the process. Thus, a function applied at the federated server as in *Listing 7.12* can be directly mapped to a routine at the data source. Note that the predicate in the where-clause refers to the BLOB column SDEBLOB carried through from the nickname to the view to prevent the constructor for the spatial types from interfering as was explained before in Section 7.4.1.

```
SELECT geom, name
FROM   n
WHERE  db2gse.EnvelopesIntersect(sdeBlob,
          -108.00, -107.00, +43.00, 44.03, 1003) = 1
```

Listing 7.12: Spatial query on nickname at federated server

The query sent by the wrapper to the foreign data server is shown in *Listing 7.13*. The SQL statement in the listing is generated by the federated server in conjunction with the wrapper. It refers directly to the base table and not any view. The data source compiles and optimizes the SQL statement independently and the invocation of the function *EnvelopesIntersect* is now considered by the data source as a valid predicate for the exploitation of an existing spatial index. Thus, the spatial predicate applied to the view over the nickname is effectively pushed down to the data source.

```
SELECT A0."GEOM", A0."NAME"
FROM "STOLZE"."T"A0
WHERE (db2gse.EnvelopesIntersect(A0."GEOM",
    -1.080000000000000e+02, -1.070000000000000e+02,
    4.300000000000000e+01, 4.403000000000000e+01, 1003)=1)
FOR READ ONLY
```

Listing 7.13: Pushed down spatial query processed at data source

7.5 Summary

We analyzed the situation with respect to the support of spatial data in federated database systems in this chapter. The SQL/MED standard [ISO03l] already considered user-defined types (UDTs), especially structured types. It includes the field CURRENT_TRANSFORM_GROUP_FOR_TYPE in the foreign-data wrapper descriptor area to inform the federated server about the transform group that shall be used to convert a (transformed) spatial value retrieved from the foreign data source to the corresponding data type at the federated server. These facilities would be sufficient if the existing federated products adhered to this infrastructure.

Today's products like the IBM WebSphere Information Integrator [IBM04c] usually do not support spatial data types. Oracle database links can handle spatial data but only if the foreign data source is also an Oracle database system and uses exactly the same definition for the spatial types as the federated server. That is not applicable in heterogeneous environments.

With the lack of support in the products, other means have to be found to provide federated access to spatial data. We presented the techniques that are built on the conversion of the spatial data to an external format, i. e. the WKB representation. The WKB had to be extended to also include the numeric identifier of the associated SRS in order to have a self-contained value.

We implemented a wrapper that provides the seamless access to spatial and non-spatial data managed in the GRASS geographic information system [GRA05]. Together with adjustments to the definition of some of the spatial functions (in particular the logic referring to the handling of the SRSs) the wrapper could detect spatial predicates and inform the DB2 federated server that those predicates are evaluated at the GRASS data source. Thus, we have proven that the push-down of spatial predicates is feasible with a small implementation effort.

GRASS as the underlying GIS does not have a coherent and consistent application programming interface (API). Therefore, we did not attempt to cover the complete spectrum of possibilities for the management of spatial data in GRASS. The wrapper was tailored to support only the native file-system based storage model and not the option to manage spatial and non-spatial data in PostgreSQL and PostGIS. As a result, the wrapper was directly impacted by the separate storage of spatial data in so-called vectors and non-spatial data accessed through the DBMI interface. The many-to-many relationship between vector elements and tuples in the DBMI databases had a substantial impact on the performance if an inadequate approach to join spatial and non-spatial data was chosen.

Other issues with GRASS were the poorly documented API. Some functions were not documented at all. Furthermore, the GIS does not scale because it tries to handle all the data in main memory. The memory management is another problematic issue in

GRASS since only the very basic and inadequate system calls `malloc` and `free` are used internally and the more advanced infrastructure mandated by the DB2 wrapper API cannot be plugged in. Likewise, the error handling in GRASS falls short of modern requirements on such a component. All that made the coding of the wrapper more complicated but did not prevent a successful solution.

The GRASS wrapper is very basic and it does not provide any cost information to the DB2 optimizer. Future work on the wrapper will focus on that, along with the support for additional spatial operations besides the function *EnvelopesIntersect*. The performance of a wrapper is always an issue and additional effort has to be invested there and in GRASS itself.

We applied the results from the GRASS wrapper to the DRDA wrapper that can be used to access DB2 data sources from a DB2 federated server. This wrapper follows the provisions of the SQL/MED standard very closely, even if it does not support spatial data. Some additional adjustments were necessary in the configuration to overcome this obstacle. We presented two different approaches to access federated spatial data. The approach relying on the *DB2_PROGRAM* actually allowed us to implement the push-down of spatial predicates.

8 Spatial Replication

Replication is a mechanism to copy data between multiple database systems, relational and also non-relational. Relationships between tables, i. e. foreign key constraints can be preserved in the process if desired. A wide variety of facilities is available to configure and schedule the replication processes using time-based or event-driven techniques. The database environment in enterprises is usually quite heterogeneous. Integrating replication products into such environments makes it mandatory that different database management systems from various vendors are supported seamlessly.

Considering that spatial data, as introduced in Chapter 2, shall be handled like any other data in a database, it is fundamental that geometry information can also be replicated between different database systems. Unfortunately, the existing replication products cannot deal with spatial data as with integers or strings. If a specific product actually supports the spatial data types in a certain database system, then the state of the art is that only homogeneous support is available. The spatial information can only be replicated between database systems from the same vendor or product family. For example, Oracle Replication [Ora05b] is able to deal with Oracle Spatial but it can only replicate spatial information from one Oracle database to another Oracle database. Other products do not even natively support spatial data in any way. An example is DB2 Replication [IBM04b]. Users of spatial data cannot use such products at all and have to come up with custom replication configurations. Such configurations can hardly reach the stable and well-supported state of dedicated replication products.

Adopting a new replication tool is usually not an option in enterprise-scale system configurations. Also, just relying on tools specialized on spatial data conversion like the Feature Manipulation Engine (FME) from Safe Software [Saf05b] is often not possible as such tools do not cover the full range of replication functionality. Thus, the task is to implement mechanisms how spatial data can be replicated using the existing products only. This feat can be accomplished by converting the spatial information into a format that is recognized and supported by the respective replication tool and then replicating the converted data. At the target system, the conversion is reversed and the proper geometries are constructed in the database. We establish the following criteria for the whole spatial replication process, when implementing such a system:

- No or only marginal additional disk space shall be required for the spatial replication. In particular, the converted geometry is not to be stored persistently in the source table before or after a replication cycle. Storing the converted data during the replication itself, for example in temporal storage, is permissable.

- The spatial data should be created in the target table on the fly, i. e. no staging should be used if it can be avoided. That will reduce disk access due to page writes and no additional logging for insert operations will be necessary.
- The impact on the transactional processing taking place at the source database should be reduced as much as possible. Any additional replication-specific work (e. g. format conversions) should be done during the replication itself.
- Data format conversion should be as fast as possible to improve the overall performance of the replication process.
- No additional administration tasks should be necessary beyond the initial setup.
- No specialized replication process shall be introduced. The replication of spatial data is to be handled with the existing architecture.

In specific situations, those points might be alleviated or even discarded in favor of other constraints. For example, if the performance of the replication has a higher priority than the transactional processing on the source or target databases, preprocessing can reduce the load on the replication cycle.

We introduce the basic concepts of replication in Section 8.1. Some of the existing replication products and technologies are presented in Section 8.2. That establishes a base line for our subsequent explanation how spatial data can be replicated solely by exploiting the object-relational functionality in the database systems and replication tools without any need to actually change the program code of those products. The techniques build on a canonical, standardized data format to represent geometric information. We explain in Section 8.3 which format is deemed to be the most suitable one. We develop and describe two strategies to replicate spatial data in Section 8.4. This section also explains how an ideal solution for spatial replication might look like in the future and which facilities are needed for that in the SQL/MM spatial standard. We close this chapter with a summary on our findings for spatial replication technologies in Section 8.5.

8.1 Basics on Replication

Replication is the process to establish and maintain two (or more) identical data sets, typically in different databases. It duplicates and copies the affected data. Replication does not have to be performed on a complete database but a subset of the tables and a subset of the data in the tables is sufficient. *Figure 8.1* illustrates the concept of replication between relational databases on an abstract level. Two database systems, possibly located on different physical machines, are connected via the replication process. The replication process retrieves the data changes from the first database system and sends

it to the second system. Replication mechanisms can usually provide the data changes to several target database systems and, thus, it is a form of a classic publish/subscribe system [Leh02].

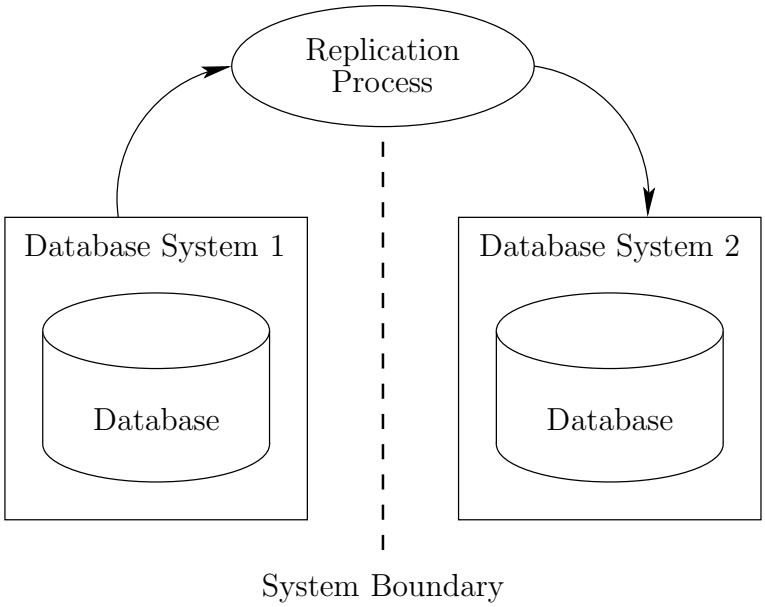


Figure 8.1: Overview of the replication process

In most cases, replication is an asynchronous process, i. e. any changes made to the data set are scanned periodically and asynchronously propagated to the copies (replicas). The replication is not performed within the same transaction that initiated the changes. Consequently, there is a gap between the changes in the replication source and the replicas. This gap is called *latency* and all relevant replication tools available today offer ways to configure this parameter.

There are many versatile application scenarios where replication is employed. One usage scenario is dedicated to increasing the availability of the data. If data is stored in several places, an outage at the primary site does not compromise the whole system as the replicas are still on-line and the transactions originally working on the primary site can be rerouted to the replicas. Thus, the probability for an outage of the whole system is significantly reduced.

Another situation concerns load balancing. Instead of installing a single server which all clients connect to and process their requests, several servers are used, possibly located in different geographical locations. Each client connects to the closest server, reducing network traffic, response times due to local network access, and also causing a distribution of the load on the database servers themselves. The databases on all servers are synchronized using replication mechanisms. This scenario is especially interesting if (monetary) costs are implied by network traffic.

Mobile computing [Gol06b] also uses replication in order to allow clients to function properly and query a local database even if no connection to the company-wide database server and network exists. A subset of the data is loaded on the client machine so that it can work independently. Insurance companies and others often adopt those means for their sales representatives. Under certain conditions, it might also be worthwhile to choose the subset of the data being replicated based on spatial constraints, i. e. the sales representatives should acquire new customers and for that the data (names, addresses) of potential customers is loaded on his/her system. The selection could be based on spatial information, e. g. the distance between the customer and the employee. We do not focus on such spatial restrictions (or other ways to select a subset of data to be replicated) in this chapter as it is straight-forward.

Yet another scenario is related to testing purposes where data from a production system is copied to a test system. For example, if a new version of the database system software is to be applied, the existing data and application need to be run with the new version in order to verify the proper functioning of all components before the new version goes live. Employing replication in such a context allows an easy synchronization of the test and verification system with the current production system without impacting the production system more than necessary by only adding the load to extract the data changes regularly.

The above listed examples are by no means complete. Many other situations can be found where replication plays a major role in the corporate IT infrastructure. The specific application is not relevant for the subsequent discussion concerning spatial data in replication scenarios. It is obvious to treat spatial data as first class citizens in a database system and also to replicate it like any other data.

Replication of data between two database systems is often done as a one-way copying, i. e. the data is replicated from a source system to a target system. That way, the target system contains all changes of the source system but it can also have additional data modifications that were performed on the target system only. Those additional modifications are not propagated back to the source system in such a uni-directional setup. Bi-directional replication is usually also supported by replication products. In such a scenario, all changes in the source system are sent to the target system and vice versa. Due to the asynchronous nature of replication, it is necessary to implement specific synchronization strategies and mechanisms for collision detection and resolution [NH02]. Otherwise, it could happen that a change is propagated from the source to the target, where it is again treated as a data change and propagated back to the source so that a loop occurs. Such strategies and collision handling are not considered any further. Instead, we focus solely on spatial data processing in replication scenarios.

Two techniques exist to actually move the relevant data when replicating data from one relational database to another. The data changes are captured and they can either be propagated to the target system using persistent message queues [IBM05], or the data

changes can be put into staging tables in a relational database system itself. At the target system, the data is retrieved from the queue or the staging table and inserted into the respective tables. A queue-based replication mechanism is shown in *Figure 8.2*. An application drives queries and data modifications (insert, update, or delete operations) against a table in the database. The data modifications need to be logged in order to ensure the recoverability in case of a crash or other failure of the source database. As a side effect, the log can be used by a Capture process to asynchronously extract the data changes. In the figure, the table has three columns named ID, C1, and C2, respectively. The data changes are written to the message queue so that they are sent through the queuing mechanism to the target system. The Apply process at the target system receives the data changes by reading the queue and applies them to the target table.

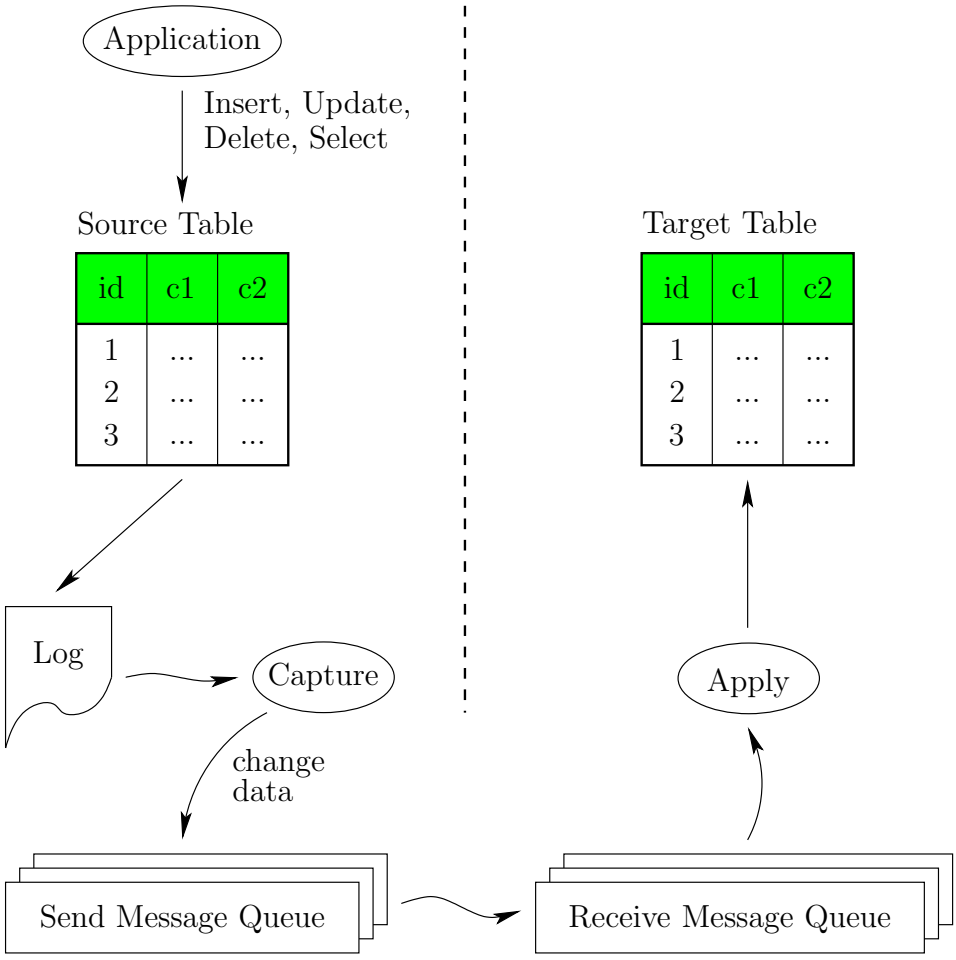


Figure 8.2: Queue-based replication

Figure 8.3 shows how staging tables are used instead of message queues. The Capture process extracts the data changes from the log written by the database system and inserts

it into a relational table in the same or another database. The staging tables typically reside in the source database itself and they are directly accessible via SQL statements. That is the primary difference to message queues, which are often implemented as tables in an RDBMS internally but are not directly accessible to other applications. Besides the changed data, some additional system-specific columns are present in staging tables to keep track of transaction boundaries and the specific operation that triggered the data modification. From there, the Apply process retrieves the changes and propagates them to tables in the target systems.

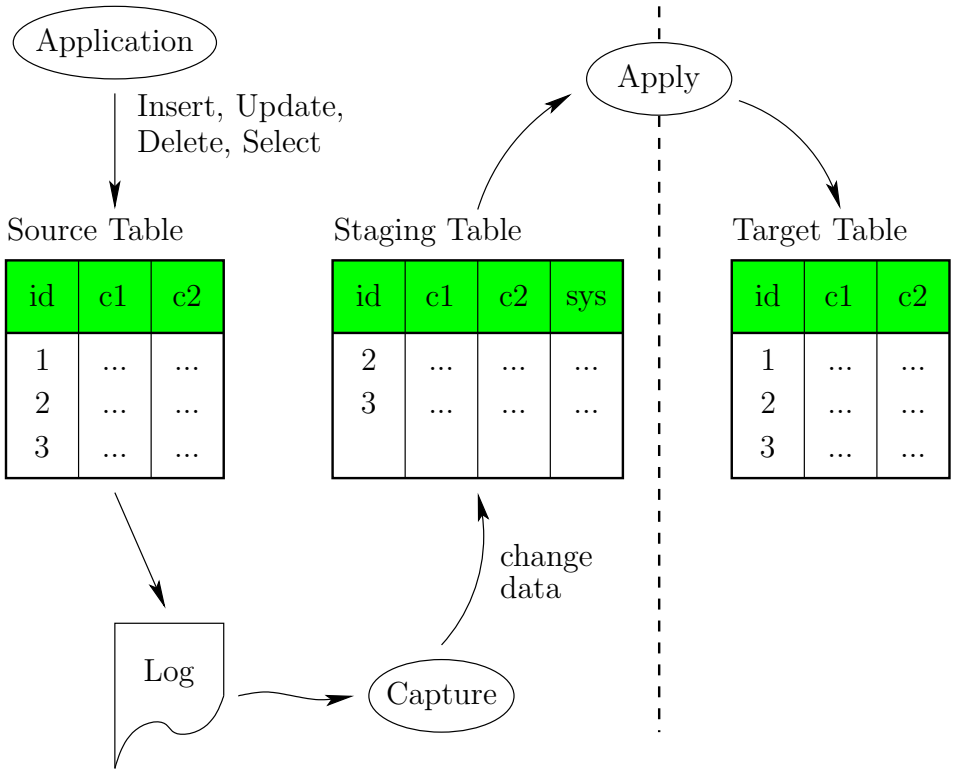


Figure 8.3: Using relational staging tables for replication

An alternative to analyzing the database system log records can be found in triggers. Some replication products use a set of triggers to intercept the insert, update, and delete operations and either fill respective information into the staging table or generate the proper messages for the replication message queue. Naturally, the triggers add load on the transactional processing, which has to execute the trigger logic. The decision whether a log-based or trigger-based capturing comes into play is dependent on the integration depth of the replication product with the respective database management system (DBMS). Internal knowledge on the structure of the database logs can be taken advantage of if both products are developed by the same vendor or if the source code of the DBMS is available. Typically, such a situation leads to a log-based implementation of the Capture mechanism.

There are several ways to apply the data changes, depending on the purpose of the target system. Regular insert, update, and delete operations could be used if the target system is operational, i. e. clients connect to that system and perform their processing on it. If the target system is just a hot stand-by for the case of an outage of the primary system, the changes could be applied directly to the database as if crash recovery or a roll-forward recovery is being performed. The second option is available only if the in-depth knowledge of the recovery mechanisms implemented in the target system are available. That can also be provided by the DBMS vendor only. A technical restriction is that the recovery-oriented Apply process has to have the complete control over the database system. That implies that no user can be working with the target database and, for example, query or modify the data stored in its tables.

8.2 Overview on Replication Products

A wide variety of replication products can be purchased today. Usually, the vendors of DBMSs provide a replication suite that is specifically tailored to their system. We introduce some of these products here and analyze them regarding their support for spatial data. Many different features that may be implemented are not described unless they are relevant to the task at hand. Further replication products exist, for example, Sybase iAnywhere [iAn04]. Since they have no relation to spatial, they are also ignored.

None of the replication tools can replicate spatial data in heterogeneous environments. Some tools can take care of spatial data in homogeneous configurations, i. e. replicating between databases managed by the same DBMS. However, that falls far short of a deep integration of spatial data in enterprise environments. Therefore, we develop the needed new techniques.

8.2.1 IBM DB2 Replication

IBM ships a replication suite, called DB2 Replication [IBM04b], as part of its DB2 Information Integrator package, since renamed to IBM WebSphere Information Integrator. The Information Integrator is tightly coupled with the DB2 Universal Database offering.

DB2 Replication is tailored to uni-directional and bi-directional replication between DB2 database systems. Non-DB2 systems are also supported if those systems can be integrated through the federated technology in the Information Integrator. For example, Oracle or Microsoft SQL Server databases can become the source and/or target in a replication setup if the tables and views to be replicated are registered as (federated) nicknames in a DB2 database. Once the proper configuration is in place, DB2 Replication itself does not have to consider the differences between the various database systems with this approach. The federated capabilities already take care of it and perform the necessary conversions and type mappings, for instance.

Initially, the replication suite consisted only of the so-called *SQL Replication*. The Capture process reads the log files maintained by DB2 during the transactional processing on the database and the change data for the replication is extracted and stored in (relational) staging tables in a database. A queue-based replication mechanism, called *Q Replication*, was added recently and now both can be used, depending on the specific requirements of the environment where the replication tools are to be deployed.

As for the supported data types, only the predefined types like INTEGER, VARCHAR, or CLOB are handled. Additionally, user-defined distinct types that are based on those predefined types can be used. Structured types are ignored by DB2 Replication as no facilities exist for them. Thus, spatial data types cannot be replicated natively and the user has to resort to other techniques like the view-based approach to hide the spatial data as is described in [Sto04].

8.2.2 IBM Informix Enterprise Replication

Data stored in databases managed by the Informix Dynamic Server (IDS) can be replicated with Informix Enterprise Replication (ER) [IFX05]. ER assumes that the source and target systems are both IDS databases.

Informix Enterprise Replication is queue-based. The change data extracted from the database logs are inserted into a message queue, shipped to the target system and read there so that the changes can be applied. Staging tables are not used, so a direct interference and adjustment of these tables as Section 8.4 lays out is not an option.

A major advantage with respect to spatial replication is the fact that ER can handle user-defined opaque types. Two functions named *streamread* and *streamwrite* have to be implemented for that. When a row that includes values of opaque types is queued for replication, ER invokes the function *streamwrite* on those opaque values to convert them from their internal to an external, unstructured representation, i. e. binary streams. Each binary stream is simply sent to the target system as-is. ER calls the function *streamread* at the target system for the inverse operation. It builds the internal representation of the value from the external one it got. This mechanisms implies that the source and the target system in the replication scenario have to use matching and complementing functions for both conversions.

Opaque types are the means for the spatial types in Informix as Section 6.1.2 explained. Spatial DataBlade implements the two functions *streamread* and *streamwrite* to directly exploit the ER infrastructure. Besides the relatively effortless replication support for spatial data, The external binary stream has another advantage for the Informix Spatial DataBlade: The internal structure of the data in the spatial types may evolve and change over time. Using a stable external format introduces a level of independence from the internal format and allows the successful replication between different version of the Spatial DataBlade module.

8.2.3 Oracle Replication

Oracle Advanced Replication (OAR) [Ora05b] is another product for the replication between homogeneous database systems. This time, the underlying databases are to be managed by an Oracle DBMS.

Messages queues, called *transaction queues*, are the means to send the change data from the respective source system to the target. The mechanisms to populate the transaction queues cover all predefined and user-defined data types, including object types. The spatial data types in Oracle Spatial are object types and, therefore, OAR can directly be used to replicate spatial data between two Oracle database systems.

Oracle Advanced Replication imposes some restrictions on the data types at the source and target systems. Namely, the object types must have exactly the same type definition. This restriction includes not only the data types of the attributes and the respective length specifications, e. g. VARCHAR2(40). Also the order of the attributes in the type definition must be identical. Another restriction is that no type hierarchies are supported. Fortunately, that is not an issue for spatial data in an Oracle database given that only the type SDO_Geometry exists and features for subtyping and inheritance are not exploited.

The restrictions for the replication of object types stem from the implementation in OAR. The replication tool is not aware of the semantics of the object type (spatial data type). Therefore, it requires a one-to-one mapping on the source and target systems in the replication setup. That way, it will (potentially recursively) decompose an object into its single attributes and send the attribute values through the transaction queues. At the target, those attributes are simply combined again into the object. As a major difference to the Informix Enterprise Replication it is to note that no single value with an external data format is employed for the objects. Obviously, that also has an impact on the replication between heterogeneous database systems given the diverse implementations that were summarized in Section 6.1.

8.2.4 StarQuest Data Replicator

The StarQuest Data Replicator (SQDR) [Sta05] is presented as a product that is not developed by a DBMS vendor. Its main benefit is that it can replicate between Oracle Database, DB2, and Microsoft SQL Server systems natively. That means, SQDR takes care of type mappings and data conversions itself and does not rely on federated capabilities (cf. Chapter 7) as does, for instance, DB2 Replication.

The capture process of SQDR does not rely on the database logs. That is due to the independence from the database vendors and that an exact and detailed documentation for the log records is usually not available. Building a product on such an unstable documentation base was not an option for SQDR. Triggers and stored procedures are

employed to capture the data changes. The changes are stored in staging tables similar to the mechanism used by DB2 SQL Replication.

SQDR only handles predefined data types like VARCHAR or SMALLINT. Large objects (LOBs) can also be replicated, but more specialized user-defined structured types or object types are not supported. Thus, the replication of spatial data using the StarQuest Data Replicator is not possible either.

8.3 Data Format for Spatial Replication

Many different data formats to encode spatial information were developed (cf. Section 2.3.3). Specialized tools exist to convert the information from any one such format to others [Saf05a]. One format has to be chosen as the canonical format for replication purposes. Ideally, the selected format is already supported by the spatial extensions that are available for relational database systems. Therefore, the list of possible formats becomes very short and only the following four data formats can be considered. Each format is either standardized by the International Organization for Standardization (ISO) or it is a de-facto industry standard.

Well-known text The well-known text (WKT) representation is standardized as part of the SQL/MM spatial standard. It represents the definition of a geometry in a textual format.

Well-known binary The second external format defined by the SQL/MM spatial standard is the well-known binary (WKB) format. It is platform-independent and specifies exactly how the structural information of a geometry along with the actual point data is to be encoded in a binary stream.

Geography Markup Language Based on the eXtensible Markup Language (XML), the Geography Markup Language (GML) [ISO05b] is a textual format. Whereas WKT uses parenthesis to separate the point lists, rings or polygons of a geometry, GML employs XML-tags for that purpose. Therefore, GML is much more verbose than WKT without providing significant additional value.

ESRI shape The shape format was originally developed by ESRI [ESR98] and not by a standardization body like ISO. However, the format has a substantial market share and is supported by virtually every spatial product.

Usually, the various spatial extensions for relational database systems provide functions that convert a geometry to the respective external format. Additionally, the products also implement functions or constructors that take the textual or binary representation as input and construct the equivalent geometry from that.

Choosing between a textual and a binary representation is simple. A textual representation is usually longer than a binary representation because more characters are needed for each floating point number that represents a single coordinate value. Another advantage of binary formats is the higher accuracy. The points in spatial data are stored as floating point numbers using the IEEE 754 standard [IEE85]. Converting such floating point values to their textual representation immediately introduces rounding issues [Gol91]. The accuracy of the data would be reduced already at the source of the replication, and the reconstruction of the binary data at the target could further decrease the accuracy. Directly transferring the binary data avoids this problem.

The previous considerations leave the WKB and the ESRI shape formats. A close observation of the ESRI shape format reveals that this format does not distinguish between multi-linestrings with only a single part, e. g. "multilinestring ((10 10, 20 20))", and primitive linestrings, e. g. "linestring (10 10, 20 20)". The same applies to multipolygons and polygons. Thus, the specific type information could be lost during the replication process. However, the argument does not apply if it is known that the spatial data has a certain specific type, e. g. if only linestrings are to be replicated. Using the proper constructor function on the target table will ensure that the correct specific type is reconstructed in such a case.

Another possible advantage of the WKB format can be found in the native support for little and big endian encoding. If the source and target system both use the same encoding, no additional conversion will be necessary. The ESRI shape format stores all information in little endian (except the header for each geometry) and therefore requires the conversion of each floating point number from big to little endian and vice versa if a big endian system is used. Of course, that has only a small impact on the performance of the conversion routines.

The WKB format has the disadvantage that it is slightly more verbose than the shape format. It requires a few more bytes to represent a geometry. Summarized, the WKB representation is usually the best choice. However, no final decision can be made for either binary format without knowing the specific environment and configuration.

8.4 Spatial Replication Strategies

Two strategies can be used to replicate spatial data using solely the functionality implemented in the available database management systems and replication products. The first approach fitting into this quite narrow corset is based on the replication of large objects (LOBs). The spatial values are converted to binary large objects (BLOBs), then replicated and converted back at the target system. However, the replication of large objects is usually dealt in a manner differing from the replication of other data types like numbers or strings. Therefore, we go into the details that are necessary to align spatial data with these mechanisms in Section 8.4.1.

Not all replication systems are able to deal with large objects. Thus, we present another strategy based on fragmenting the external representation of a geometry in Section 8.4.2. Each fragment is a short (binary) string that is replicated like the traditional predefined data types. The fragments have to be generated at the source system and recombined at the target system so that the spatial value can be constructed there.

An additional third strategy applies to point data only. A point is not a very complex object and it consists of just two coordinates – as opposed to all other kinds of geometries. Thus, a point value can be replicated by extracting those coordinate values, storing them in additional columns of the table. The replication process itself ignores the spatial column but does include the two additional columns. Triggers can be used at the target system to create the respective geometry again. This technique cannot be applied to curve and surface geometries because those would have to be handled point by point. That results in long runtimes, not to mention the need to extract all the single points from the spatial values. For points, the approach is rather obvious and we do not delve into it in more detail.

The spatial types are usually implemented as user-defined types. The techniques demonstrated here were specifically developed to handle spatial data, but they can also be applied to other user-defined data types. In Section 8.4.3 we suggest an implementation that should be followed in the replication tools in order to replicate spatial data as well as data of other user-defined data types.

8.4.1 Replicating via Binary Large Objects

The initial idea to replicate spatial data is to convert each geometry to a BLOB value and transfer it as such to the target system. Usually, replication tools do support BLOBs. The remaining question is how to perform the replication adhering to the criteria declared before, i. e. how to avoid materialization of the BLOBs and to reduce the impact on the regular transactional processing.

The approach chosen by Stolze [Sto04] relies on views on top of the base tables that are to be replicated. A view over the source tables establishes the conversion of the spatial data to its WKB representation via the function *ST_AsBinary*, and a view over the target table implements the reverse transformation, i. e. the construction of the spatial values in the target system. This scheme is illustrated in *Figure 8.4*.

Staging Tables

The replication between views is especially an option if staging tables are used to hold the data to be replicated. DB2 SQL Replication [IBM04b] is one product that provides this technology. Other products like the StarQuest Data Replicator (SQDR) [Sta05] adopted a similar approach. Typically, the developers of replications products made a conscious

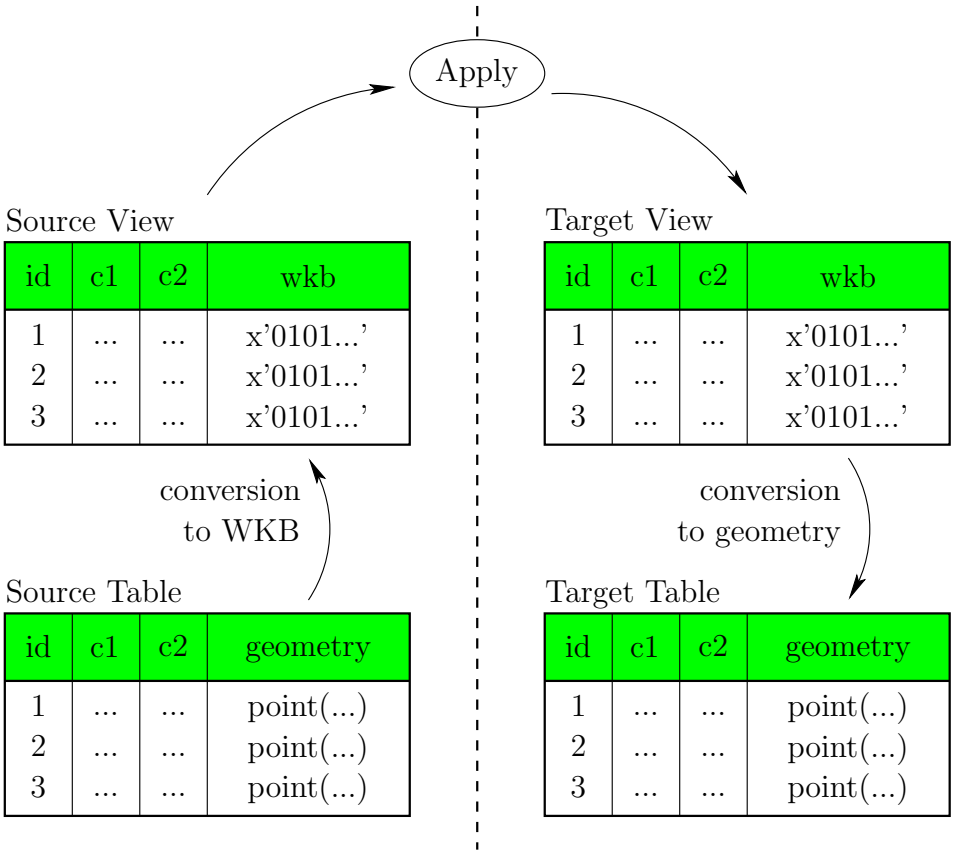


Figure 8.4: Replicating spatial data via BLOBs

decision not to store any changed LOB data in the staging tables (called *change data* tables by IBM) for performance reasons. Instead a Boolean update-indicator is used in the staging table to reflect whether the LOB itself changed or not. For example, an update-indicator of type CHARACTER(1) is set to 'U' if the LOB changed. The Apply process grabs the actual LOB data from the source table in that case. The benefit is to save disk space and time by avoiding the copying. All other changed attributes (non-LOBs) are retrieved from the staging table only.

If a view SV is used to deal with the spatial data in the source table ST, the view has to be prepared accordingly. Additionally, the update indicator needs to be maintained in the staging table for ST as well the staging table for SV. *Figure 8.5* depicts the logic on the source side of the replication scenario including both staging tables. The staging tables are marked yellow to distinguish them from the actual user tables and views. The staging table for SV is itself just a view, and the replication tool derives its definition from the `CREATE VIEW` statement used for SV. The assumption is that the view SV is based on ST and, thus, capturing changes on the base table automatically captures the changes for SV as well if the same logic that is applied on the data in the base table is also applied on the captured changes in the staging table.

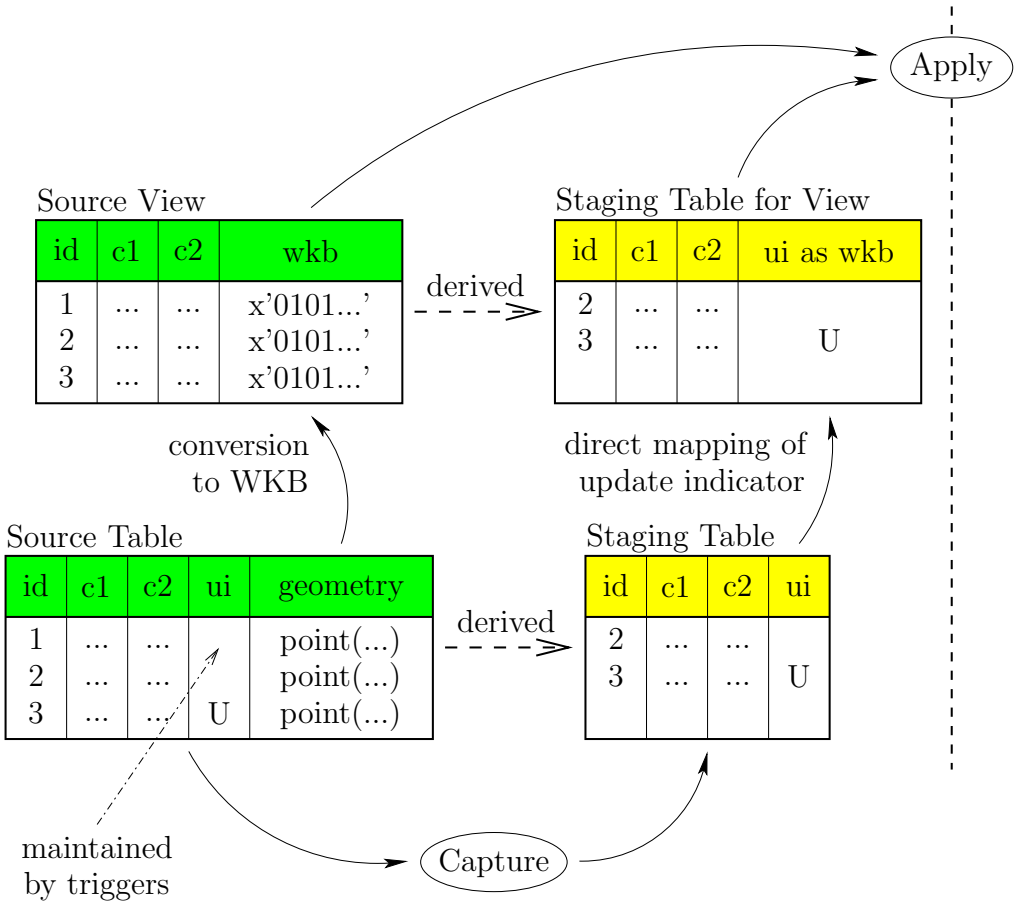


Figure 8.5: Staging tables for spatial replication using BLOBs

We add the update indicator column to the base table along with triggers to maintain it transparently. If and only if the value in the spatial column changes, a trigger sets the value in the update indicator; otherwise, the indicator is reset. The regular Capture processing picks up the changes on the update indicator column and stores its modification in the staging table. The final piece on the source side is to ensure that the update indicator, which is so far treated like a regular column, is considered to be an update indicator for the BLOB column in SV. Thus, the relationship between the BLOB column and the update indicator in the staging table of the view needs to be established. DB2 SQL Replication defines such a relationship by using the same column names, for instance. The details to define such a setup for a source table can be found in [Sto04].

Summarized, the view-based replication adds a very small overhead for the update indicator column. The transformation of the source geometry is completely handled when the Apply process queries the staging tables and accesses the view SV to retrieve (what it assumes to be) BLOB values. A query against the view causes the *ST_AsBinary* method to be invoked and the geometries are converted to their WKB on the fly.

The target table TT can be handled in a similar fashion. A view TV is defined over the target table, and that view will be used as target for the Apply process. With TV being defined in the same way as SV, the data types and names of the columns in the source and the target views simply line up.

The issue is that the inclusion of the spatial function *ST_AsBinary* in the view definition immediately results in the view being non-updatable, i. e. read-only. The DBMS does not know automatically what the reverse mapping between a BLOB and a geometry shall be. The introduction of instead-of triggers in the SQL standard [ISO03i] and also in (some) DBMSs offers an easy way to provide this information to the DBMS. *Figure 8.6* shows the conceptual approach for that. When querying the view, the function *ST_AsBinary* is involved to convert the geometry to its WKB representation. An instead-of trigger kicks in if data shall be inserted into the view. The instead-of trigger discards the original insert operation and executes the trigger body instead, which converts the WKB representation to the corresponding geometry using the proper spatial constructor function.

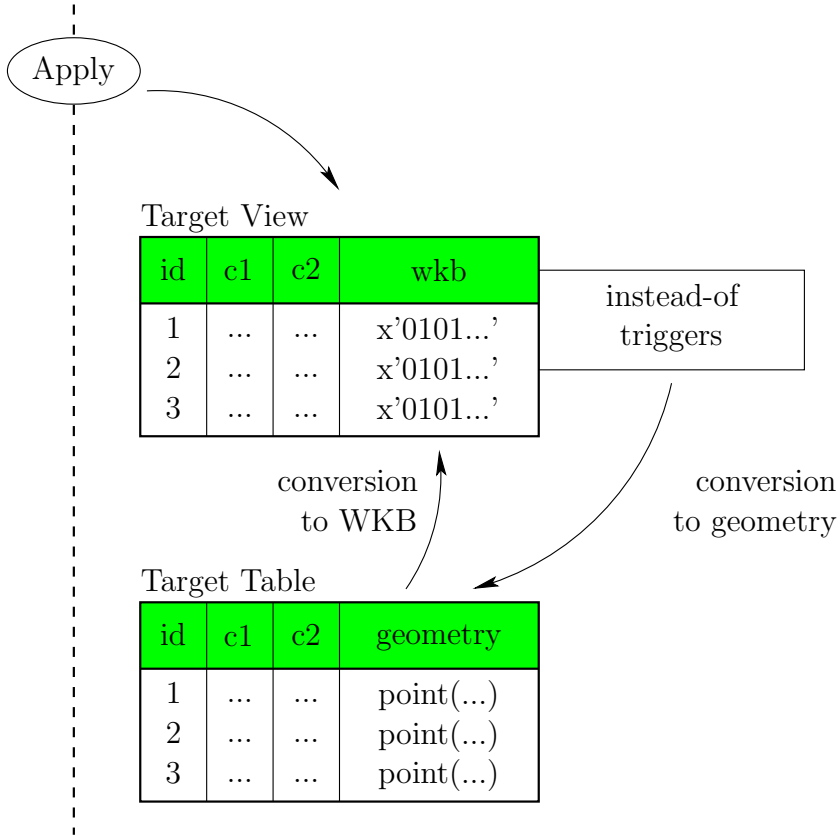


Figure 8.6: Handling target table for spatial replication using BLOBs

Queue-based Replication

Integrating the support for spatial data in queue-based replication scenarios can be handled similar to staging tables. The major difference is that no views in the relational database can be used for an implicit conversion of the geometries. The Capture process reads the database logs, extracts the data changes from there and places appropriate messages directly into the replication message queue. No access to the database is made, so the geometries cannot be converted to their WKB representation under the cover. It is not possible to give the replication tool the impression that it is working with regular BLOBs.

The consequence is that triggers on the source table ST are needed to take care of the conversion to WKB and insert the BLOB directly into the replication queue as *Figure 8.7* depicts. As a side effect, the workload for the conversion itself will be added to the transactional processing on the source database.

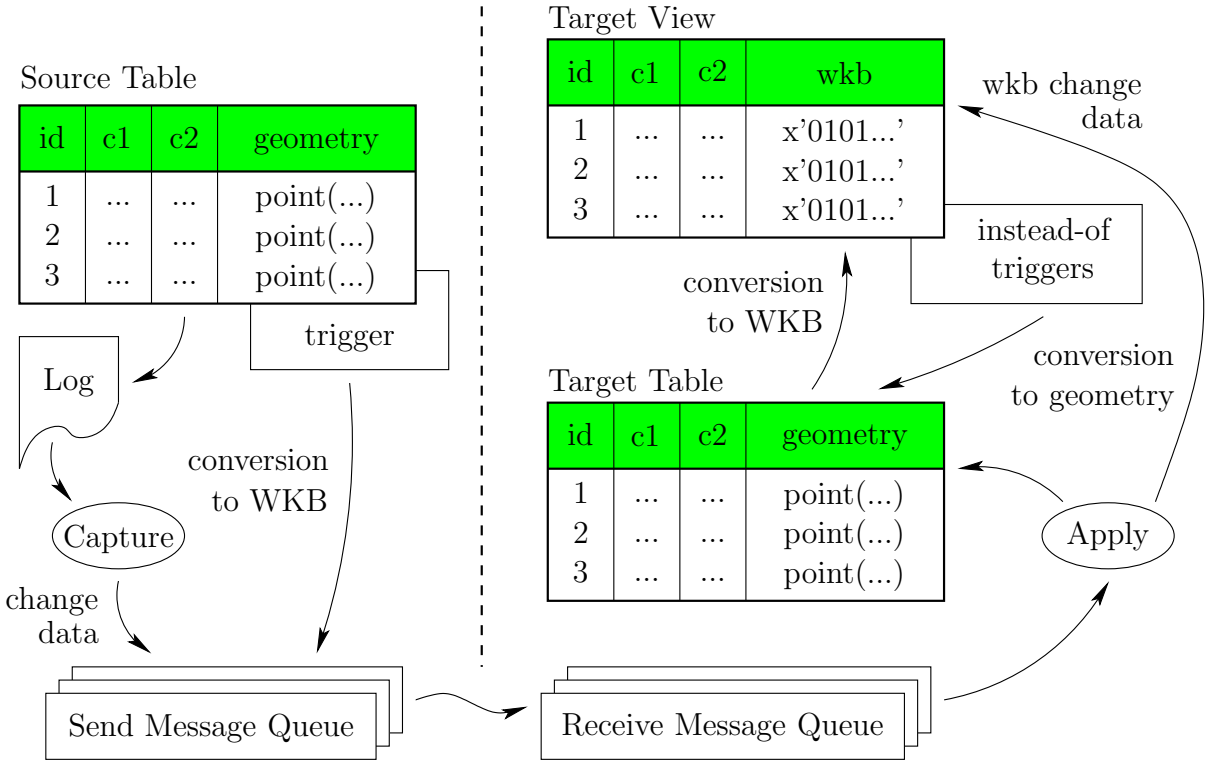


Figure 8.7: Queue-based replication of spatial data using BLOBs

The timing of the replication process in this scheme becomes a critical issue. Usually, the logs are read asynchronously. However, the BLOBs containing the WKB are inserted into the queue synchronously. Thus, a race condition exists where the BLOB is written to the queue before the actual insert or update operation that triggered the change of the BLOB in the first place.

The Apply process at the target system will have to deal with such situations so that the changes are applied correctly. The message with the data changes for the WKB has to be delayed until the data changes for the remainder of the row appears at the target system. That will require a change in the logic for the Apply process and, thus, it is not an option in general as either the source code for the replication tool might not be available and even if it is, such changes are not desirable, compared to native support for spatial data. Therefore, we develop other techniques.

A way out is the use of a separate message queue. The BLOB is inserted into another, parallel queue. Once the regular data changes are processed at the target system, the additional queue is consulted and the BLOB extracted from there. The price for such a setup is the higher administrative overhead for the parallel queues where the new one has to be managed manually because the replication tool itself is not aware of it. Additionally, there is no logic available in the replication tool that deals with the BLOB unless the Apply process considers the queue as a valid replication queue and assumes that it is correctly filled by a Capture process at the source system.

A better approach builds on a temporary but logged table. When a change of the spatial data at the source table occurs, a trigger creates the WKB and inserts it into the temporary table. The insert operation triggers a log record to be written, and the Capture process will read the log record in a timely manner. If the trigger fires after the insert or update operation on the source table, the BLOB update is scheduled after it as well and the Apply process will receive the messages from the queue in the proper order. Although this approach solves the timing issue, its costs lie in the additional storage required for the temporary table. The temporary table needs to be pruned regularly. Pruning must be done in such a way that any deletions from the side table are not propagated to the target systems in order to prevent the spatial data from being deleted there.

Heterogeneous Environments

Carrying the concepts to heterogenous environments comes with another layer of complexity unless the replication tool like SQDR [Sta05] can handle diverse database management systems. The data types and structures in the heterogeneous systems have to be mapped to each other. An example for BLOB-based spatial replication in a heterogeneous database environment is evaluated in [Het05]. The spatial and non-spatial data stored in an Informix database is copied to a DB2 database. Federated technologies provide the link between the different database management systems. However, none of the available federated products does support spatial data in heterogenous environments as we explained in Section 7.2. Therefore, the approach for the replication of the spatial data either involves the techniques presented in Chapter 7 as far as they are applicable or, alternatively, the spatial data is converted to its well-known binary representation already at the source system, i. e. at the foreign data source. With respect to spatial

replication both approaches are very similar. The basic difference is where the conversion from the spatial data to the WKB BLOB is made: at the federated server or the foreign data source.

The architecture to combine federated technology and BLOB-based spatial replication is shown in *Figure 8.8*. The source table resides in a different database system. The figure presents the by far most common situation that spatial federation is not available and the foreign data source has to take care of the conversion to the WKB. Thus, the source view and its staging table reside at the foreign data source and the nicknames at the federated server (marked in cyan) refer to those views via the database link. It is also possible to place the target table in yet another database system. In that case, another nickname enters the picture at the target side. Instead of inserting into the target view, a nickname that refers to the true target view will be used.

Even though the wrapper employed by Hetterle [Het05] does support BLOBs and DB2 SQL Replication also supports BLOBs in homogeneous scenarios, the replication from an Informix database to a DB2 database does not allow the replication of BLOBs. That is an unnecessary restriction imposed by the replication tool. Therefore, Hetterle proposed a work-around to close this gap.

First, a source table is created in the DB2 database. This table contains a BLOB column and it is registered as replication source. Since it is a DB2 table, the replication of BLOBs is supported. A second step replaces this table and its associated staging table with nicknames pointing to the view and its staging table in the Informix database, but the replication setup information is not changed. The final step involves the regular checking performed by the Apply process. It detects that the source table is not a base table any longer but rather a nickname and, therefore, the replication for its data is stopped. The results of this particular check must be reset using a trigger on the replication catalog tables. More details can be found in [Het05].

8.4.2 Fragmenting the Spatial Data

Depending on the used replication and/or federation products, it might not be possible to resort to BLOBs that contain the converted geometries in their well-known binary representation. Reasons could be that large objects are not supported at all, BLOBs are read-only in the respective federated product, or the performance of BLOBs access would be deemed to be insufficient. If a replication tool builds on top of such a restrained federated gateway, it inherits the restriction when replicating between heterogeneous database systems. To still be able to handle spatial data in such contexts, another approach must be adopted where BLOBs are completely avoided.

The idea we propose makes use of the situation that replication always transfers all changes that belong to a single transaction together. In other words, multiple pieces of data are transferred from the source database to the target database in a single

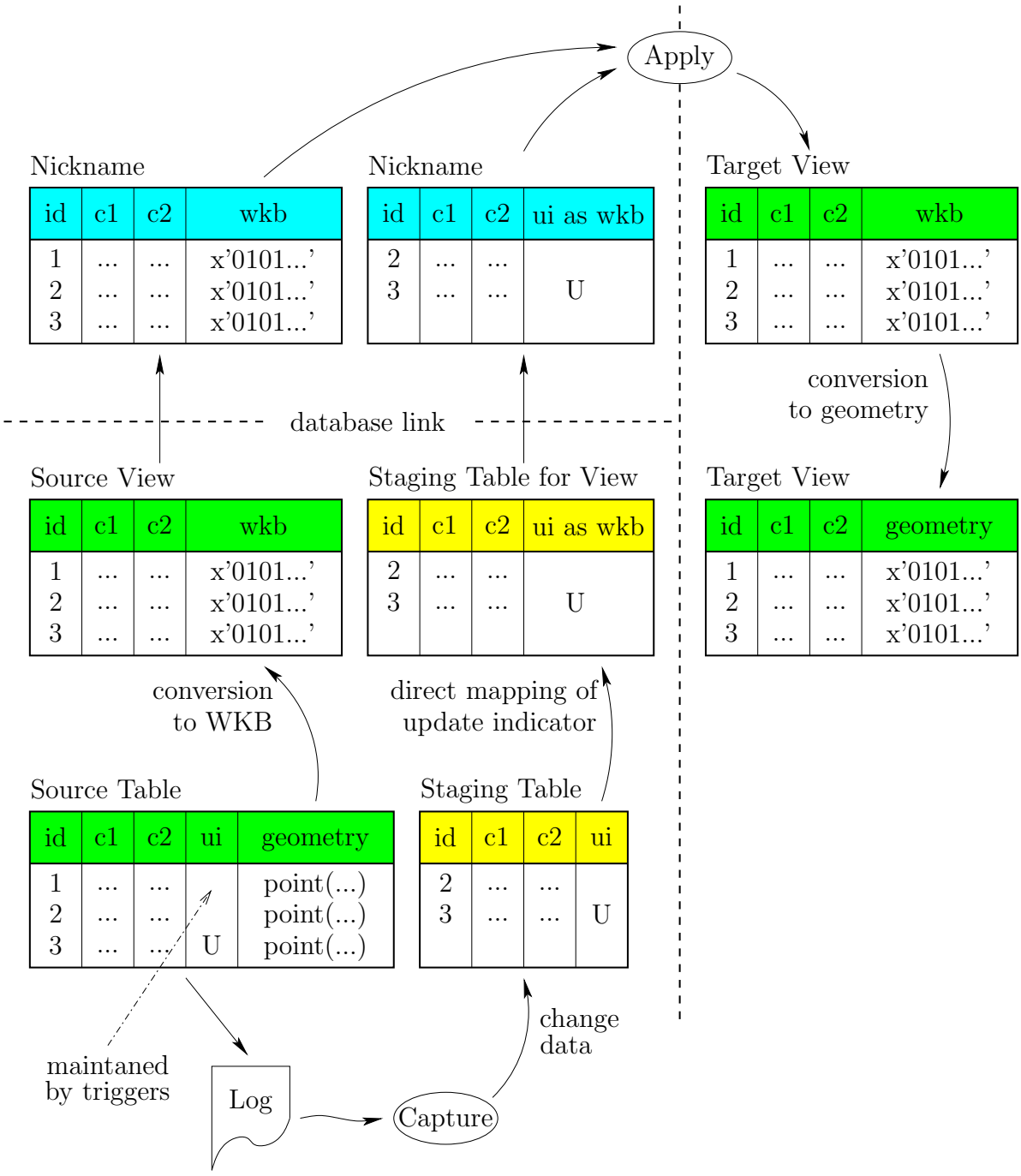


Figure 8.8: Heterogeneous BLOB-based spatial replication

replication unit. Furthermore, the changes to several tables in the source system can be grouped into so-called *Replicate Sets* or *Subscription Groups*.[1] That grouping of changes on the same or multiple source tables implies several SQL insert operations being performed at the target system.

If the replication products are already tailored to processing multiple pieces of data, the geometries themselves can be chopped into fragments and transferred that way. The effect is that the well-known binary or well-known text representation of a geometry is handled as a series of CHARACTER values or binary variations of that type. The fragments are all sent as changes in a single unit and, thus, applied together at the target.

A successful replication of spatial and non-spatial data using fragmentation between an Oracle Database and a DB2 system is described in [Sal05]. Although the replication itself is feasible, the main issues are the criteria declared initially: the load on the transactional processing on the source database shall not be increased, a simple maintainability is to be guaranteed, and any additional work should be placed on the replication process only if possible. Most of these criteria are not met by Salomon [Sal05].

Several options for the spatial replication without BLOBs are discussed. First, a view is used to convert the geometry to its WKT representation using a CHARACTER VARYING column very much like the BLOB-based replication described above. A second approach materializes the geometry WKT in an additional column of the source table and keeps it synchronized with the spatial column via triggers. However, variable length strings in today's database systems have length limitations that are quickly exceeded by the spatial WKT format if the geometries become more complex. For example, DB2's VARCHAR values must not be longer than 32,000 bytes, and Oracle Database even sets the limit at 4,000 characters. However, a shapefile describing the boundary of the United States of America using a single multi-polygon geometry with more than 200,000 points at a sufficiently high resolution and precision would never fit into such a short string.

The third and final approach developed by [Sal05] makes use of an additional table, a so-called *side table*. The side table contains multiple rows for a single geometry in the source table. Each row contains only a fragment of the geometry WKT representation, and all fragments together form the complete WKT. The side table circumvents the length restrictions originating from the usage of the character data types. Unfortunately, it does not fully comply with the intention to keep the overhead for the transactional processing on the source database small. Furthermore, the data in the side table is indeed materialized, so that additional disk space is needed. Also, the WKT representation is not the best choice for spatial data as was discussed before.

The situation can be improved to better comply with the initially established criteria by using a view as side table to generate the fragmented WKB representation on the fly. The view will represent all geometries of the source table in their well-known binary

[1] The various replication products choose different naming conventions for the same concept.

representation, split into short chunks that fit into a CHARACTER VARYING column, optionally using a binary variation of the data type if that is available. The content of that fragment view are then replicated. That establishes the means to effectively replicate the spatial data together with the data in the source table without relying on LOBs. *Figure 8.9* demonstrates such a configuration at the source system. The Apply process will read both staging tables and transfer the change data in both to the respective target system.

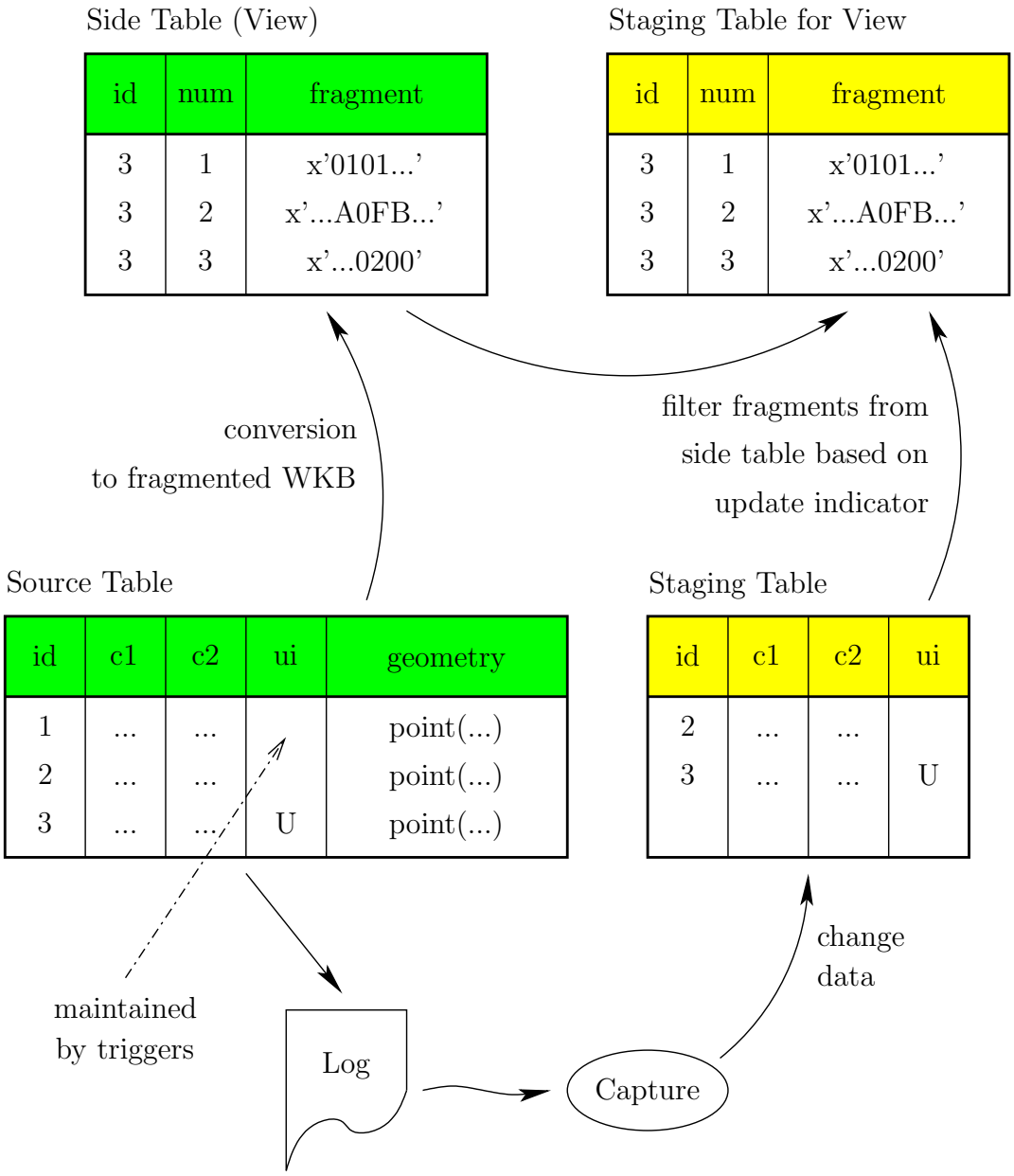


Figure 8.9: Fragmenting WKB representation of geometries

Given the very different structure between the source table and the fragment view, the corresponding staging tables have to be very different as well. That becomes a particular issue as the staging tables only contain those rows that had data changes applied to them. The staging table for the source table ST is straight-forward. It only deals with the changes on the non-spatial column. The staging table for the fragment view SV, however, requires special considerations. This staging table should only reflect the current changes and not all the rows from SV. Thus, a means to filter the rows from the side table (and enhancing them with the required replication-specific system columns) is necessary. Furthermore, replicating string data implies that the string data is stored in the staging table itself. Thus, the staging table for SV has to be defined as a view itself to avoid additional disk usage and also to prevent increased load on the transactional processing on the source database.

With the usage of a view, the required filtering can be accomplished using an update indicator as in the BLOB-based replication scenario. Here, the indicator will not serve as an update indicator for a BLOB column but rather as a filter criteria only. It is not replicated to the target database. The information provided by the update indicator could also be kept in a separate table, together with the key values of the changed row and the mandatory replication-specific system columns required for a staging table. Such an additional table would need to be maintained by triggers defined on the source table. Resorting to the update indicator stored directly in the source table results in a simpler implementation, however.

The fragmentation logic for the WKB representation is provided by a user-defined function (UDF) that produces a (temporary) table as its result. Each row in that table is comprised of the identifier for the current WKB fragment and the fragment itself. The total number of fragments could be returned as well so that it can be used for error detection purposes at the target system.

The resulting temporary table is first employed in the definition of the fragment view SV that acts as replication source. The second place for the function is in the corresponding staging table (view). The definition of such a table function is shown at the beginning of *Listing 8.1*. Depending on the source database management system and its set of available functions, it may be necessary to implement an additional function that performs the logic of the function *SUBSTR* on binary strings, i. e. it takes a BLOB as input and returns a piece of it as a binary character string. The shown UDF generates fragments that are not longer than 4000 bytes. Of course, this limit could be changed or parameterized. The second SQL statement in the listing demonstrates how the fragment view SV itself is defined. The last SQL statement creates the corresponding staging table for SV as a view, combining the data from SV and the staging table for ST. The restriction to represent only those rows where a geometry changed is implemented here. The replication-specific system columns in the staging table for SV are derived from the staging table for ST and only sketched in the statement by setting them in italics.

```
CREATE FUNCTION fragmentWkb( wkb BLOB )
   RETURNS TABLE ( fragmentNumber INTEGER,
      fragment VARCHAR(4000) FOR BIT DATA )
   LANGUAGE SQL
   RETURN
      WITH RECURSIVE t(cnt, fragment) AS (
         VALUES ( 0, NULL )
         UNION ALL
         SELECT cnt+1, SUBSTR(wkb, cnt * 4000,
                  CASE
                     WHEN LENGTH(wkb) > cnt * 4000
                     THEN 4000
                     ELSE LENGTH(wkb) - (cnt-1) * 4000
                  END)
         FROM   t
         WHERE  cnt * 4000 < LENGTH(wkb) )
      SELECT cnt, fragment
      FROM   t
      WHERE  cnt > 0

CREATE VIEW v AS
   SELECT t.id, f.fragmentNumber, f.fragment
   FROM   t, TABLE ( fragmentWkb( t.wkb ) ) AS f

CREATE VIEW sv
   SELECT st.id, v.fragmentNumber, v.fragment,
          st.system-columns
   FROM   v JOIN st ON v.id = st.id
   WHERE  st.ui = 'U'
```

Listing 8.1: Setup for source system to replicate fragmented WKB

The source system of the replication configuration adheres to the criteria established initially if the implementation resorts to fragment views to handle spatial data. In particular, the following three aspects are covered, that were not available in [Sal05]:

1. No additional work has to be performed during the transactional processing to convert the geometries to the WKB representation and chop the WKB into smaller fragments (except the necessary triggers to maintain the update indicator column).
2. No additional disk space is needed as the WKB is not materialized.
3. No administrative tasks are necessary beyond the initial setup because no side tables need to be pruned regularly.

The replicated spatial data and the other data from the same row can become out of sync because the latest value for the spatial data is consulted during the replication cycle, but the other values are the ones from the staging table. For example, a row in the source table is updated and a value v_1 is set for a non-spatial column. The spatial value is updated at the same time to a geometry g_1, causing the update indicator to be set in the staging table. Then the replication cycle is started and while this cycle runs, the same row is updated again and the non-spatial value is changed to v_2. The spatial value in the source table is also modified and now contains the geometry g_2. Once Apply reaches the row in question, it retrieves v_1 from the staging table and notices the update indicator for the spatial column. The fragment view is consulted and geometry g_2 (instead of g_1) is retrieved. Thus, the target table will contain a row that never existed that way in the source table – at least until the next replication cycle is run.

This situation is not considered as a serious issue in the industry, given that the replication of large objects is handled in the same fashion for several years already [Sta05, IBM04b]. The performance won by avoiding the copying of the LOB data outweighs the potential and temporary inconsistencies. Furthermore, today's applications for spatial data do not (yet) have such a high update frequency, making the issue less likely to appear. Also, the chance of the temporary inconsistency can be reduced by adopting a smaller interval between two replication cycles. The cost is a higher workload of the replication process. Nevertheless, the potential for inconsistencies and the additional overhead implied by another replication cycle cannot be denied.

Materializing the WKB fragments in the target database cannot be avoided completely. At least temporary storage is needed during the replication cycle. Oftentimes it is unknown how exactly the replicated data is added to the target system, e. g. new data could be inserted using a single statement for all the rows belonging to the same table, or several insert operations could deal with those rows. In the latter case, the order of the fragments in not necessarily sequentially. Thus, the scope of a single statement is not guaranteed and the access to the WKB fragments has to be established in such a context. The only option is, as *Figure 8.10* depicts, to rely on a base table that receives all fragments for a single geometry during a replication cycle. A stored procedure is invoked at the end of the cycle. This procedure performs post-processing steps to combine the fragments together and generate the new or modified geometry value in the actual target table. Additionally, the fragments are removed from the side table after the geometry is constructed. That keeps the overhead for the additional disk space low. Such a procedure was already used by [Sal05] for the replication of spatial data from an Oracle database to a DB2 database.

All major replication products offer ways to run post-processing logic at the end of a replication cycle. Thus, the integration of the spatial data into the target table can be accomplished. In summary, the logic at the target system does require additional disk space, but that disk space is only needed temporarily during the replication processing itself. After the cycle is completed, no additional storage is occupied any longer.

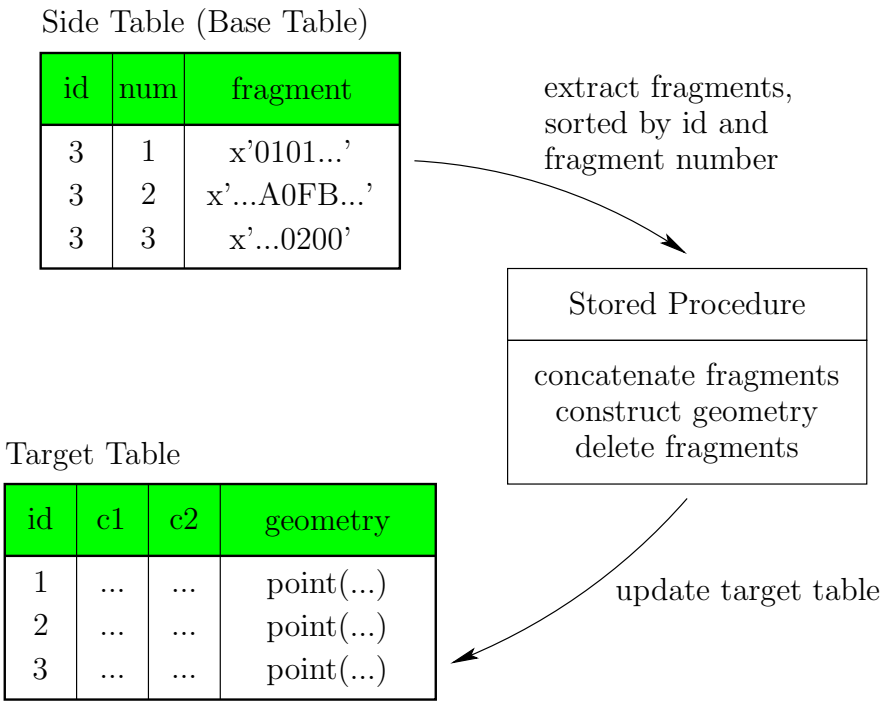


Figure 8.10: Handling fragmented WKB at target system

8.4.3 Goal for Heterogeneous Spatial Replication

The strategies described in the Sections 8.4.1 and 8.4.2 can be used to replicate spatial data. While both strategies implement the desired results from a functional point of view, the usability and setup is rather complex. Supporting tools to manage the configurations are desirable. A better alternative would be that the replication tools themselves implement the necessary logic and hide any spatial-specific additional work from the user or administrator.

A step into this direction is made by the Informix Enterprise Replication product [IFX05], which defines a very general infrastructure with its two functions *streamread* and *streamwrite* to create a value of a user-defined type from an external format or to convert it to that format, respectively. The way over such an external format is clearly better than Oracle's approach that relies on identical type definitions in the source and target systems, which is a requirement that simply cannot be fulfilled in heterogeneous environments.

The external format used by the Informix Spatial DataBlade is not documented, and it does not build on any of the external data formats defined in the SQL/MM spatial standard. Homogeneous replication tools can continue to rely on an internal format for such purposes. For non-homogeneous replication tools like SQDR [Sta05], it is mandatory that a common and standardized format exists to transfer data between source and

target systems in a well-defined manner. Essentially, it is the same issue as if an application communicates with a spatial database system and from the DBMS perspective, the replication tool is just another application. The same issues that were already discussed in Chapter 3 are applicable, namely that a fully self-contained external format is needed.

Besides the external format, the replication tools have to be aware of the fact when they are dealing with spatial data because the spatial reference system (SRS) are usually managed differently at the systems involved in the replication scenario. The replication tool has to take care of the mapping between the SRS at the source and target systems. The mapping itself can be directly implemented in the replication suite as Section 6.2 laid out. For persistent storage, it may be beneficial to hold the mapping in the replication catalog tables that describe which target table receives the data from which source table. Then the Apply process has to parse the external format created at the source system and replace the numeric identifier of the SRS with the matching identifier used at the target system. Thus, the geometry can be created immediately in the target table using the correct SRS without interference by an administrator or any hard-coded values as [Sto04] required in the definition of the instead-of triggers on the target views.

8.5 Summary

The situation with respect to the replication of spatial data is far from optimal in today's products. If the respective replication tool does support spatial data at all, this support only covers homogeneous environments where the source and the target databases are both managed by the same DBMS. For example, spatial replication between two Oracle databases is possible, replication between a Microsoft SQL Server system and an Informix IDS database cannot be handled natively.

Most replication products provide the facilities to replicate binary large objects. That can be exploited for the spatial replication by converting the geometries to an external data format like the well-known binary representation and then replicating this representation. As we have shown in this chapter, the BLOB-based replication is feasible and different implementations have proven the practicability. However, a special setup is required at the source and the target databases for this technique. We also like to point out that only the functional aspects were considered and performance questions still remain to be answered.

In case that BLOBs cannot be replicated with the chosen replication product, a similar approach can be adopted. The spatial data is still converted to an external format, but this time the resulting representation is fragmented into several pieces that fit into strings of type CHARACTER VARYING, for example. The replication of strings is a basic functionality that every acceptable replication product does support. While the fragmentation at the source system is straight-forward and can be implemented with

views, the reconstruction of the spatial value at the target system is more involved as the fragments have first to be collected and cached in a table. Once all the pieces arrived, the reconstruction can be initiated at the end of a replication cycle.

Both techniques, the BLOB-based replication and the fragmentation, have in common that no additional permanent disk space is needed to store the data in a different format. We kept the impact on the transactional processing at the source system to a minimum if staging tables are employed to hold the data changes. In that case, the conversion to the external data format (and possible fragmentation of the result) can be loaded on the actual replication cycle, in particular the Apply process, with the adoption of views. That does not hold true for queue-based replication where the conversion itself has to be done as part of the transactional processing since the external representation of the geometries is to be sent to the replication queues directly. The administrative overhead for the configuration is smaller and the setup is more robust for the BLOB-based replication. Therefore, we recommended the adoption of the BLOB-based approach over the fragmentation if possible.

The strategies for the replication of spatial data are not restricted to geometries. Values of any user-defined data type can be replicated as we described, as long as the necessary conversion functions are provided. Thus, the developed techniques are more general.

Unfortunately, there is no SQL standard dedicated to define the infrastructure and functions for replication. Such a standard would be very desirable as it would be a good place to define the self-contained external data format and the mentioned functions. With that, replication of spatial data in heterogeneous database environments could someday be as seamless as the replication of integer numbers or strings is today.

The SQL/MM spatial standard should at least define a binary external data format that is self-contained by including for each geometry the numeric identifier of the SRS or the SRS itself. With such a format and the associated routines to convert geometries to it (or vice versa) being available in the spatial products, the replication tools could begin to support the heterogeneous spatial replication.

Part IV

Summary

9 Conclusions and Outlook

In this closing chapter of the thesis we summarize the findings in the different areas for spatial data support in relational database environments. Additionally, new questions are raised and an outlook on possible future work is given.

9.1 Summary of Results

The previous chapters described functionality that is defined by the SQL/MM spatial standard [ISO03d] and we identified a number of gaps in the standard. Closing those gaps is an essential task for the further adoption of spatial technology in non-GIS areas.

The available spatial extensions for relational database systems implement dedicated data types and a set of associated methods to construct geometry values, store them in relational tables, and to perform basic spatial operations. A working group of the International Organization for Standardization (ISO) is tasked with the standardization of this kind of functionality. Chapter 2 gave an introduction to the SQL/MM spatial standard as well as important products in this field. According to the standard, geometries are restricted to 2D objects in two-dimensional data space. Advanced features like improved language bindings, graph functionality, and 3D data types and operations are still missing. We identified the most important areas where functionality should be added to the standard. Additionally, we analyzed distributed database environment regarding their support of spatial data in federation and replication scenarios.

Language Bindings

Writing applications that leverage spatial functionality in an RDBMS is not an easy task. The communication of geometries between application and database has to be achieved by means of byte streams that encode geometries in an external, standardized format like well-known text (WKT) or well-known binary (WKB). Thus, a spatial object is converted by the application to its WKB, for example. Then it is passed on to the database system where a function is invoked to construct the corresponding geometry value. The external representation is also needed in the opposite direction, i. e. to retrieve a geometry from the database. The application is responsible to parse it and build a spatial object. In short, a direct use of spatial objects is not supported in language bindings like JDBC and SQLJ. Spatial data is not treated as a first class citizen

as, for example, integers or strings. However, that is very much desirable to support the same spatial operators in the application as in the relational database management system (RDBMS).

We explained in Chapter 3 how to improve this state. A spatial class hierarchy is provided as a means to represent geometries in a Java application. Then the JDBC classes were extended to directly work with such spatial objects. For that, only three methods were required to retrieve spatial objects from result sets or `OUT` parameters of stored procedures and to associate them with a parameter marker in a prepared statement. We proposed the addition of the methods *getGeometry* and *setGeometry* to the classes *ResultSet*, *PreparedStatement*, and *CallableStatement*.

The full set of spatial operations can be made available to applications through this spatial class hierarchy. The respective methods can either be implemented natively in Java (and existing packages like JTS already do so) or the spatial database system is consulted each time through an SQL query. Our performance measurements have shown that referring to the database system comes with a penalty due to the internal communication overhead. Java code is slower than compiled C/C++ code so that the spatial extension may compensate this overhead in spatial calculations. An automatic decision between both approaches provides the best overall performance and functionality.

Sun Microsystems, as organization that controls the JDBC specification, will have to add the new functionality to this document. It is not in the domain of the SQL/MM spatial standard to define such enhancements. As for SQLJ [ISO03f], the situation is different because this standard is developed and maintained by ISO. We suggest to change the SQL/MM spatial standard based on the results in this thesis.

Graph Integration

Geometries and topological information are tightly related if multiple geometries are considered. Spatial databases usually store a large number of geometries. A relational table holding the streets of a certain region like a state or country is a typical example with several thousand linestrings. This spatial data can be transformed into a graph so that graph operations like shortest paths or minimum spanning trees can be applied. Performing such operations directly at the DBMS level offers powerful functionality to applications because results of graph operations can be further processed with SQL.

The current SQL/MM spatial standard does not acknowledge that and it does not provide any graph-related operations. The upcoming version of the standard will (most likely) include a function *ST_ShortestPath*. However, this function does not fit seamlessly into the spatial relational world. It is insufficient and impractical to use since it does not return the linestrings comprising exactly the shortest path but rather a set of values that identify the (full) linestrings in the function input. We proposed a more general approach to integrate graph functionality with spatial data in Chapter 4.

We based the interface to graphs solely on geometries. That means, geometries form the input to construct the graph and geometries comprise the results of graph operations. The benefit for applications using this infrastructure is that no specialized post-processing is necessary. In the end, we treated the graph functionality as a kind of spatial index mechanisms. The usage of graphs is completely transparent and we avoid the explicit representation of graphs in the database.

We explained the mapping of geometries to graphs and vice versa. Generally, points that are used to define a spatial value are converted to vertices in the graph. Lines connecting those points become edges. This scheme is directly applicable to linestrings. Points and polygons require background information from the application to properly construct a graph. Especially the information for the definition of edges is needed. We presented several possible approaches for that.

A variation of the *ST_ShortestPath* function was described in Section 4.4.5. It should replace the function that is in the current working draft of the SQL/MM spatial standard because of its improved usability. Additionally, the SQL/MM spatial standard should incorporate the methods like *ST_DistanceToPoint* and *ST_ExtractSegment* that we proposed. Those routines are needed for the reverse mapping from graphs to linestrings.

We implemented the concepts in the Spatial Graph Extender. That allowed the evaluation of the ideas as well as measurements and comparison of different algorithms for the graph construction. Although the Java-based implementation can surely be improved, the performance is already in an acceptable range.

The prototypical implementation of the Spatial Graph Extender has shown that a direct integration of spatial data with graphs in a state-of-the-art RDBMS is not only possible functionality-wise but also offers acceptable performance. The only problematic areas of the extender were the weak integration into transactional contexts and the main memory storage of graphs. These issues could be addressed with a direct integration of graph functionality into the database engine.

Support for 3D Data

The SQL/MM spatial standard only considers points, linestrings, and polygons (and collections thereof) in the two-dimensional data space. Support of a third and even fourth dimension are added in the current working draft, but this mechanism is only a means to store additional information in the geometries but not to exploit those values for spatial computations. Nevertheless, today's applications often do need a true 3D data space and also 3D geometries. The best examples are computer-aided design (CAD) systems.

In Chapter 5 we explained in detail what is necessary in the standard to fully support 3D geometric data and operations. The extension of the spatial type hierarchy is very

light-weight. We introduced the new types ST_Solid and ST_Polyhedron as geometric primitives. The types ST_MultiSolid and ST_MultiPolyhedron complement the primitives with their respective homogeneous collections. The existing spatial functionality can be directly used with the new types. Naturally, the algorithms have to be adjusted, but the SQL interface remained stable. We added a limited set of methods applicable only to the new types. For a seamless integration and backward-compatibility to the current standard, it was necessary to manage 2D and 3D operations at the same time. The spatial reference system (SRS) that is associated with each geometry can be consulted to reliably distinguish between traditional 2D operations (favored by GIS applications) and calculations in $\mathbb{R}^3$.

The implementation of the 3D Extender has demonstrated that it is possible to integrate 3D data types into the DB2 Spatial Extender [IBM04d]. The necessary modifications were limited to the registration of the spatial functions in the database. No adjustments to the external code of the extender had to be made. The new logic could be added via DB2's extensibility mechanisms like user-defined functions (UDFs). In the end, the combination of the DB2 Spatial Extender with the 3D functionality formed the 3D Extender. CGAL was used in the 3D Extender to implement spatial operations. We have shown that CGAL can provide the necessary functionality, but it could benefit from additional performance improvements. A native integration into the DB2 Spatial Extender (or respective other spatial extensions) is preferable over the work-arounds that were adopted to deal with the fact that the extender source code is not publicly available.

Spatial Federation

Another area that is impacted by spatial data is the management of complex environments that can be found in companies and enterprises. Typically, a database system is not operated as a standalone system. It is connected either directly or through applications with other database systems. Federation is an option to combine data residing in different database systems. As Chapter 7 demonstrated, the access to federated spatial data turns out to be even more complicated than replication due to the lack of support in the federated products. We summarized that the SQL/MED standard [ISO03l] actually defines the necessary infrastructure with the specification of transform groups to handle structured (and spatial) data. However, existing products that implement wrappers for relational and non-relational data sources do not yet consider spatial data.

Simple work-arounds for spatial federation, like employing views to hide the fact that spatial data is involved, cannot provide a proper solution. It is not possible to push-down spatial predicates in queries to the remote data source since the federated server is not even aware of the spatial data. Furthermore, none of the external spatial data formats is fully self-contained with respect to SRSs. SRSs have to be managed manually in such systems.

The only solution is to modify the wrappers to become spatially aware. By means of an example, a wrapper for the GRASS GIS [GRA05] has been implemented from scratch. With the help of this wrapper, we have shown that the push-down of spatial predicates could be achieved without requiring any modification internal to the database system that acts as federated server. The handling of the SRS identifiers and the mapping of them could also be integrated into the wrapper. The performance of the GRASS wrapper turned out to be acceptable. The approach we have chosen for the integration was sufficiently efficient – as was DB2's federated framework. GRASS itself is not very well implemented, however.

The knowledge gained from the GRASS wrapper was subsequently applied to the existing DRDA wrapper that connects a DB2 federated server to another DB2 database. We achieved the push-down of spatial predicates to the data source with some changes to the definition of the spatial functions.

Spatial Replication

Replication tools are another connecting mechanism in distributed environments. Given that the installed database systems may be from different vendors or product families, the replication products have to be able to operate in homogeneous and heterogeneous environments. That imposes serious issues when spatial data is involved because the implementations of the spatial extensions vary widely as Chapter 6 illustrated.

The SQL/MM spatial standard gives much leeway regarding the actual implementation. Therefore, the only viable option to replicate spatial data between database systems is to resort to an external representation like WKT or WKB. We explained in detail in Chapter 8 how spatial replication can be setup without requiring any change to the spatial extensions or the replication tools. The basis for these restrictions is that production environments typically do not tolerate prototypes and build on supported products instead. Thus, either large objects (LOBs) can be replicated as-is, or the LOB is fragmented via views into small pieces and then transferred like regular string or binary data. The described configurations are not very friendly to the administrator and native support in the replication tools is desirable. Nevertheless, we designed the concepts in such a way that they have a minimal impact on the transactional processing taking place at the source of the replication setup. Additionally, not only geometries can be replicated but values of any user-defined type (UDT) can be treated in the same way.

Replication is not covered by any part of the SQL standard and also not by any part of SQL/MM. The replication process is merely an application from the perspective of the DBMS. Therefore, it does not merit any standardization. Given the importance of replication in enterprises, it should be considered that the SQL/MM spatial standard is improved by defining a common, standardized mechanism for spatial replication.

9.2 Future Work

This thesis discussed some important aspects and features for spatial data handling that are not yet incorporated into the SQL/MM spatial standard. These features should be realized as proposed in the products. Or rather, the products should implement the concepts in a way that fits best with their architecture. Either way, special attention will have to be paid on performance and scalability questions.

The features that we described comprise a list of the most important topics for spatial data processing in the context of relational database systems. However, there are further technologies that should be considered to incorporate spatial functionality in the future. Some possibilities are given below.

Data Mining

As was initially explained, spatial data is becoming more and more common these days. That leads to the situation that not only experts in the field of geographic information system (GIS) deal with it, but also users who have different knowledge and generally less spatial expertise. They do not only want to store spatial data in their database systems, they also want to exploit it and extract any explicit or hidden information from it like it is done for traditional, non-spatial data for years already, i. e. to mine the data. As long as only a small group of expert users was concerned with spatial data, there was no real pressure to develop automatisms and products for that.

There are many examples for spatial data mining. Mining the data collected by emergency systems allows the identification of areas with a high crime or accident rate while simultaneously considering other factors like the average household income or ethnic background in the areas. A sales company can employ data mining to support business decisions regarding the future development of the company [Nie05]. Knowledge about customers and branches can be combined (including location information) to identify potential places for a new branch to be build.

Data mining (DM) and knowledge discovery in databases (KDD) are well-established fields for analyzing data and discovering new connections in the available data pool [HK04, ZK02]. Part 6 of the SQL/MM standard [ISO02] is dedicated to the mechanisms for data mining on relational data. Despite it being in the SQL/MM series, this standard does not have any cross references to other parts. In particular, spatial data is not considered. This situation does not only apply to the SQL/MM data mining standard but also to products, even if some progress has been made there recently [Ora05a]. The majority of the currently available data mining products that handle relational data are not aware of spatial data. Likewise, existing spatial mining tools do not operate on relational systems and use their own storage mechanisms. Thus, the DBMS user looses a major source of potentially important information by ignoring the spatial data [Hen05].

Raster Data

Traditional GIS products can not only manage vector data but also provide mechanisms to add raster or image-based data. Typical examples are satellite imagery and areal photographs (remote sensoring). The properties of raster data are quite different from vector data. A region on the Earth's surface is covered by a raster. Each cell contains some information, e. g. color or temperature values. Raster data provides a complete space partitioning of the affected region. The amount of data managed in a single raster depends on the image resolution, i. e. the size of each cell. A higher resolution (smaller cells) gives more fine-grained data at the expense of higher storage costs, which may reach several GB for a single image. A detailed analysis of the requirements and implementation details for raster management in an RDBMS can be found in [Pag06] where image pyramids, multiple bands, and georeferencing are considered.

Various GIS products that are built on top of relational database systems already contain modules to manage raster data. Additionally, some database vendors developed extensions specifically for it [Ora05d]. Thus, there is a strong argument for consolidating and standardizing such extensions as part of the SQL/MM spatial standard. Unfortunately, the SQL/MM spatial standard ignores raster data completely until now. Even the definitions in the SQL/MM still image standard [ISO03e] are not used. One reason may be that the functionality of this standard is not very adequate for raster images due to the high image sizes of rasters [Sto01].

Another image and spatial related field of application can be found in the analysis of image content. Depending on the image, spatial functionality may be beneficial to extract features from it and to store them separately as geometries. An example for that is given in [Ign01] where chemical molecules are photographed and subsequently analyzed regarding the spatial relationships of specific elements and groups in the molecules. Tying this functionality back to raster data, real-world features like streets or houses could be extracted from raster images as well in an automated fashion.

Feature extraction exceeds the current functionality of image extensions that are available for the various database systems [Sto02]. Both, the SQL/MM spatial standard and the SQL/MM still image standard, will have to be aligned and extended for that.

Content Management

Relational databases are the means to manage structured data. However, very often the data is only semi-structured or not structured at all. Content management systems (CMSs) are tailored to semi-structured and unstructured information [Boi04]. Contrary to an RDBMS, a CMS is not concerned with tables but rather with documents. The documents can consist of sections, paragraphs, sentences, etc. The actual information, like the name and address of a customer, is embedded in the content of a document.

Additionally, the contents may comprise different types of media like text or images. CMSs store the documents in such a fashion that searches across many documents for certain content can be accomplished easily and efficiently.

CMSs are not spatially aware. That means, if the documents contain some sort of spatial information, e. g. addresses, then the spatial properties are completely ignored and only the text of the address is relevant. However, it may be worthwhile to search across documents by exploiting the spatial information. For example, correlations between documents describing events in a certain region may be of interest.

In order to build a spatially aware content management system, the CMS has to be extended to allow the storage of explicit spatial data in its internal structures. Another alternative is the addition of mechanisms to identify and extract spatial information implicitly from a document, e. g. the search for addresses and the geocoding of those addresses to geographic points. Furthermore, spatial functionality has to be propagated through the CMS API to applications to exploit the spatial capabilities.

9.3 Final Remarks

It can be said that the integration of spatial data in enterprise database environments is very relevant for many applications. The integration carries requirements that are not yet addressed by the SQL/MM spatial standard. Also, the products that implement this standard today cover these aspects only partly or not at all so that an adoption of spatial technology is not considered by many users. Some of the concepts required by applications are not yet developed, making this area to an interesting research topic.

This thesis contributed solutions for 3D and graph functionality as well as a better integration of spatial data into language bindings, federated system configurations, and replication setups. The full support for spatial data in applications does not stop there, however. Future directions were outlined and a lot of work still has to be done.

Appendices

A SQL/MM Spatial Information Schema

All views in the SQL/MM spatial information schema are direct mappings of the underlying base tables in the spatial definition schema. The only exception to that is the view ST_SPATIAL_REFERENCE_SYSTEMS. The base tables are – although not mandatory for conformance to the SQL/MM spatial standard – shown as well in the following listings to provide a complete and self-containing definition of the information schema.

The elements in the SQL listings that are set in italics identify implementation-defined meta-variables. The values for those variables must be defined and documented by any product that implements the SQL/MM spatial standard. The values have to be reflected in the ST_SIZINGS view so that applications can retrieve this information and react accordingly.

A.1 ST_SPATIAL_REFERENCE_SYSTEMS

```
CREATE VIEW st_informtn_schema.st_spatial_reference_systems AS
   SELECT srs_name, srs_id, organization,
          organization_coordsys_id, definition, description
   FROM   st_definition_schema.st_spatial_reference_systems

CREATE TABLE st_definition_schema.st_spatial_reference_systems (
   srs_name CHARACTER VARYING(ST_MaxSRSNameLength) NOT NULL,
   srs_id INTEGER NOT NULL,
   organization CHARACTER VARYING(ST_MaxOrganizationNameLength),
   organization_coordsys_id INTEGER,
   definition CHARACTER VARYING(ST_MaxSRSDefinitionLength) NOT NULL,
   description CHARACTER VARYING(ST_MaxDescriptionLength),
   CONSTRAINT st_srs_name_primary_key PRIMARY KEY(srs_name),
   CONSTRAINT srs_id_unique UNIQUE (srs_id),
   CONSTRAINT organization_null CHECK (
      ( organization IS NULL AND
        organization_coordsys_id IS NULL ) OR
      ( organization IS NOT NULL AND
        organization_coordsys_id IS NOT NULL ) ),
```

```
   CONSTRAINT organization_unique CHECK (
      ( organization IS NULL AND
        organization_coordsys_id IS NULL ) OR
        ( 1 = ( SELECT COUNT(*)
                FROM   st_definition_schema.a
                          st_spatial_reference_systems AS t
                WHERE  t.organization = organization AND
                       t.organization_coordsys_id =
                          organization_coordsys_id ) ) ) )
```

A.2 ST_UNITS_OF_MEASURE

```
CREATE VIEW st_informtn_schema.st_units_of_measure AS
   SELECT unit_name, unit_type, conversion_factor, description
   FROM   st_definition_schema.st_units_of_measure

CREATE TABLE st_definition_schema.st_units_of_measure (
   unit_name CHARACTER VARYING(ST_MaxUnitNameLength) NOT NULL,
   unit_type CHARACTER VARYING(ST_MaxUnitTypeLength) NOT NULL,
   conversion_factor DOUBLE PRECISION NOT NULL,
   description CHARACTER VARYING(ST_MaxDescriptionLength),

   CONSTRAINT st_units_primary_key PRIMARY KEY(unit_name),
   CONSTRAINT unit_type_value
      CHECK ( unit_type IN ( 'ANGULAR', 'LINEAR' ) ),
   CONSTRAINT factor_value CHECK ( conversion_factor > 0.0 ) )
```

A.3 ST_SIZINGS

```
CREATE VIEW st_informtn_schema.st_sizings AS
   SELECT variable_name, supported_value, description
   FROM   st_definition_schema.st_sizings

CREATE TABLE st_definition_schema.st_sizings (
   variable_name CHARACTER VARYING(ST_MaxVariableNameLength)
                    NOT NULL,
   supported_value INTEGER,
   description CHARACTER VARYING(ST_MaxDescriptionLength),
   CONSTRAINT st_sizings_primary_key
      PRIMARY KEY ( variable_name ) )
```

A.4 ST_GEOMETRY_COLUMNS

```
CREATE VIEW st_informtn_schema.st_geometry_columns AS
   WITH RECURSIVE types(type_catalog, type_schema, type_name) AS
      ( VALUES (ST_TypeCatalogName, ST_TypeSchemaName, 'ST_GEOMETRY')
        UNION ALL
        SELECT h.user_defined_type_catalog,
               h.user_defined_type_schema,
               h.user_defined_type_name
        FROM  information_schema.direct_supertypes AS h JOIN
              types AS t ON
              ( h.supertype_catalog = t.type_catalog AND
                h.supertype_schema = t.type_schema AND
                h.supertype_name = t.type_name ) )
   SELECT c.table_catalog, c.table_schema, c.table_name,
          c.column_name, g.srs_name,
          ( SELECT s.srs_id
            FROM   st_definition_schema.
                      st_spatial_reference_systems AS s
            WHERE  s.srs_name = g.srs_name ) AS srs_id
   FROM   information_schema.columns AS c LEFT OUTER JOIN
          st_definition_schema.st_geometry_columns AS g ON
          ( c.table_catalog = g.table_catalog AND
            c.table_schema = g.table_schema AND
            c.table_name = g.table_name AND
            c.column_name = g.column_name )
   WHERE  ( c.udt_catalog, c.udt_schema, c.udt_name ) IN
             ( SELECT type_catalog, type_schema, type_name
               FROM   types )

CREATE TABLE st_definition_schema.st_geometry_columns (
   table_catalog INFORMATION_SCHEMA.SQL_IDENTIFIER NOT NULL,
   table_schema INFORMATION_SCHEMA.SQL_IDENTIFIER NOT NULL,
   table_name INFORMATION_SCHEMA.SQL_IDENTIFIER NOT NULL,
   column_name INFORMATION_SCHEMA.SQL_IDENTIFIER NOT NULL,
   srs_name CHARACTER VARYING(ST_MaxSRSNameLength),
   CONSTRAINT st_geometry_columns_primary_key PRIMARY KEY(
      table_catalog, table_schema, table_name, column_name),
   CONSTRAINT srs_supported FOREIGN KEY(srs_name)
      REFERENCES st_spatial_reference_systems(srs_name),
   CONSTRAINT column_exists FOREIGN KEY(table_catalog,
         table_schema, table_name, column_name)
      REFERENCES information_schema.columns(table_catalog,
         table_schema, table_name, column_name) )
```

B Data Formats with 3D Support

The SQL/MM spatial standard specifies two stream-based data formats for the representation of geometries. The two formats well-known text (WKT) and well-known binary (WKB) are extended in Chapter 5 to be able to carry the information for three-dimensional objects, i. e. polyhedra and multi-polyhedra. The following definitions include not only Z coordinates but also consider the presence of Z and M coordinates.

The following extended BNF notations identify the necessary enhancements for both formats. The elements and symbols that are already defined in the SQL/MM spatial standard are not repeated unless necessary for consistency reasons.

B.1 Extended Well-Known Text Representation

<multipolyhedron text representation> ::=
MULTIPOLYHEDRON *<3d zm>* *<multipolyhedron text>*

<multipolyhedron text> ::=
EMPTY
| *<left paren>* *<polyhedron text body>*
{ *<comma>* *<polyhedron text body>* }... *<right paren>*

<polyhedron text representation> ::=
POLYHEDRON *<3d zm>* *<polyhedron text>*

<polyhedron text> ::=
EMPTY
| *<polyhedron text body>*

<polyhedron text body> ::=
<left paren> *<shell text>* { *<comma>* *<shell text>* }... *<right paren>*

<shell text> ::=
<left paren> *<facet text>* { *<comma>* *<facet text>* }... *<right paren>*

<*facet text*> ::=
    <*left paren*> <*ring text*> { <*comma*> <*ring text*> }... <*right paren*>

<*ring text*> ::=
    <*left paren*> <*pointz text*> { <*comma*> <*pointz text*> }... <*right paren*>

<*pointz text*> ::= <*x*> <*y*> <*z*> [<*m*>]
<*3d zm*> ::= Z | ZM
<*x*> ::= <*number*>
<*y*> ::= <*number*>
<*z*> ::= <*number*>
<*m*> ::= <*number*>

B.2 Extended Well-Known Binary Representation

<*well−known binary representation*> ::=
        <*well−known binary representation*>
    | <*well−knownz binary representation*>
    | <*well−knownm binary representation*>
    | <*well−knownzm binary representation*>

<*well−knownz binary representation*> ::=
        <*pointz binary representation*>
    | <*curvez binary representation*>
    | <*surfacez binary representation*>
    | <*solidz binary representation*>
    | <*collectionz binary representation*>

<*collectionz binary representation*> ::=
        <*multipointz binary representation*>
    | <*multicurvez binary representation*>
    | <*multisurfacez binary representation*>
    | <*multisolidz binary representation*>
    | <*geometrycollectionz binary representation*>

<*multisolidz binary representation*> ::=
    <*multipolyhedronz binary representation*>

<*multipolyhedronz binary representation*> ::=
    <*byte order*> <*wkbmultipolyhedronz*>
        [<*num*> <*polyhedronz binary representation*>]

<*collectionzm binary representation*> ::=
    <*multipointzm binary representation*>
  | <*multicurvezm binary representation*>
  | <*multisurfacezm binary representation*>
  | <*multisolidzm binary representation*>
  | <*geometrycollectionzm binary representation*>

<*multisolidzm binary representation*> ::=
  <*multipolyhedronzm binary representation*>

<*multipolyhedronzm binary representation*> ::=
  <*byte order*> <*wkbmultipolyhedronzm*>
    [<*num*> <*polyhedronzm binary representation*>]

<*solidz binary representation*> ::=
  <*polyhedronz binary representation*>

<*polyhedronz binary representation*> ::=
  <*byte order*> <*wkbpolyhedronz*>
    [<*num*> <*shellz binary*>...]

<*shellz binary*> ::= <*num*> <*facetz binary*>...
<*facetz binary*> ::= <*num*> <*ringz binary*>...
<*ringz binary*> ::= <*num*> <*pointz binary*>...

<*well−knownzm binary representation*> ::=
    <*pointzm binary representation*>
  | <*curvezm binary representation*>
  | <*surfacezm binary representation*>
  | <*solidzm binary representation*>
  | <*collectionzm binary representation*>

<*solidzm binary representation*> ::=
  <*polyhedronzm binary representation*>

<*polyhedronzm binary representation*> ::=
  <*byte order*> <*wkbpolyhedronzm*>
    [<*num*> <*shellzm binary*>...]

<*shellzm binary*> ::= <*num*> <*facetzm binary*>...
<*facetzm binary*> ::= <*num*> <*ringzm binary*>...
<*ringzm binary*> ::= <*num*> <*pointzm binary*>...

B.3 Index-Based Well-Known Text Representation

<multipolyhedron index text representation> ::=
MULTIPOLYHEDRON *<3d zm>* *<multipolyhedron index text>*

<multipolyhedron index text> ::=
EMPTY
| *<left paren>* *<polyhedron index text body>*
{ *<comma>* *<polyhedron index text body>* }... *<right paren>*

<polyhedronz index text representation> ::=
POLYHEDRON *<3d zm>* INDEX *<polyhedron index text>*

<polyhedron index text> ::=
EMPTY
| *<polyhedron index text body>*

<polyhedron index text body> ::=
<left paren> *<point index list>* *<shell index text>*
{ *<comma>* *<shell index text>* }... *<right paren>*

<point index list > ::=
<left paren> *<pointz text>*
{ *<comma>* *<pointz text>* }... *<right paren>*

<shell index text > ::=
<left paren> *<facet index text>*
{ *<comma>* *<facet index text>* }... *<right paren>*

<facet index text > ::=
<left paren> *<ring index text>*
{ *<comma>* *<ring index text>* }... *<right paren>*

<ring index text> ::=
<left paren> *<index>*
{ *<comma>* *<index>* }... *<right paren>*

<index> ::= *<unsigned integer>*

B.4 Index-Based Well-Known Binary Representation

<polyhedronz index binary representation> ::=
    <byte order> *<wkbpolyhedronindexz>*
        [*<num>* *<wkbpointz binary>*...
          <num> *<shellz index binary>*]

<shellz index binary> ::= *<num>* *<facetz index binary>*...
<facetz index binary> ::= *<num>* *<ringz index binary>*...
<ringz index binary> ::= *<num>* *<index>*...

<polyhedronzm index binary representation> ::=
    <byte order> *<wkbpolyhedronindexzm>*
        [*<num>* *<wkbpointzm binary>*...
          <num> *<shellzm index binary>*]

<shellzm index binary> ::= *<num>* *<facetzm index binary>*...
<facetzm index binary> ::= *<num>* *<ringzm index binary>*...
<ringzm index binary> ::= *<num>* *<index>*...

<index> ::= *<uint32>*

C Acronyms

API application programming interface

BI business intelligence

BLOB binary large object

BNF Backus-Naur Form

CAD computer-aided design

CAM computer-aided manufacturing

CLI Call-Level Interface

CLOB character large object

CMS content management system

DBMS database management system

DCEL doubly connected edge list

DM data mining

DML data modification language

DMS degrees, minutes, and seconds

EPSG European Petrol Survey Group

GIS geographic information system

GML Geography Markup Language

GPS Global Positioning System

IS International Standard

ISO International Organization for Standardization

IT	information technology
JDBC	Java Database Connectivity
JVM	Java Virtual Machine
LOB	large object
LBS	location-based services
MBB	minimum bounding box
MBR	minimum bounding rectangle
ODBC	Open Database Connectivity
OGC	Open Geospatial Consortium
OID	object identifier
OR	object-relational
RDBMS	relational database management system
RPD	remote push-down
SDK	software development kit
SFS	OpenGIS Simple Features Specification for SQL
SQL	structured query language
SRS	spatial reference system
SVG	Scalable Vector Graphics
UDF	user-defined function
UDT	user-defined type
UML	Unified Modeling Language
WKB	well-known binary
WKT	well-known text
XML	eXtensible Markup Language

Bibliography

[ABC+02] B. Aditya, G. Bhalotia, S. Chakrabarti, A. Hulgeri, Nakhe. C., Parag, and S. Sudarshan. BANKS – Browsing and Keyword Searching in Relational Databases. In *VLDB 2002, Proceedings of 28th International Conference on Very Large Data Bases*, pages 1083–1086, Hong Kong, China, 2002. IEEE Computer Society. Available from: `http://www.vldb.org/conf/2002/S33P11.pdf`. 84

[AJ94] R. Agrawal and H. V. Jagadish. Algorithms for Searching Massive Graphs. *IEEE Transactions on Knowledge and Data Engineering*, 6(2):225–238, 1994. 101

[Bal00] H. Balzert. *Lehrbuch der Softwaretechnik*. Spektrum Akademischer Verlag, Heidelberg, Germany, 2nd edition, 2000. (in German). 42, 186

[Ben75] J. L. Bentley. Multidimensional Binary Search Trees used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975. Available from: `http://portal.acm.org/ft_gateway.cfm?id=361007&type=pdf&CFID=78813746`. 171

[BES+05] C. Brochhaus, J. Enderle, A. Schlosser, T. Seidl, and K. Stolze. Integrating the Relational Interval Tree into IBM's DB2 Universal Database Server. In *BTW 2005, Datenbanksysteme in Business, Technologie und Web, Tagungsband der 11. BTW-Konferenz*, volume 65 of *Lecture Notes in Informatics*, pages 67–86, Karlsruhe, Germany, 2005. Available from: `http://dx.doi.org/10.1007/s00450-005-0207-7`. 169

[Bil99] R. Bill. *Grundlagen der Geo-Informationssysteme*. Wichmann, Karlsruhe, Germany, 2nd edition, 1999. (in German). 4

[Bit05] R. Bittner. Entwurf und Implementierung einer 3D-Erweiterung zum DB2 Spatial Extender. Master's thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 151, 156, 159, 166, 171, 172

[BKOS00] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*. Springer-Verlag, New York, NY, USA, 2nd edition, 2000. 97, 130, 131, 152

[BM98] P. A. Burrough and R. McDonnell. *Principles of Geographical Information Systems*. Oxford University Press, Oxford, UK, 1998. 77

[Boi04] B. Boiko. *Content Management Bible*. Wiley, Indianapolis, IN, USA, 2nd edition, 2004. 287

[Brä05] A. Bräu. Föderierter Zugriff auf räumliche Daten in GRASS mittels DB2 UDB. Master's thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 222, 225, 228, 240

[Bri90] E. Brisson. *Representation of d-Dimensional Geometric Objects*. PhD thesis, University of Washington, Seattle, WA, USA, 1990. 129

[BRJ05] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison-Wesley, Boston, MA, USA, 2nd edition, 2005. 64

[BS95] C. W. Brown and B. J. Shepherd. *Graphics File Formats – Reference and Guide*. Manning Publications, Greenwich, CT, USA, 1995. 9, 146, 225

[Büh04] J. Bühler. Verwaltung von Geodaten in der digitalen Bibliothek MyCoRe. Master's thesis, Cartography and GIS Group, University of Greifswald, Germany, 2004. (in German). 31

[Bur04] Bureau of Transportation Statistics. BTS — Dynamap/1000 Street [online]. 2004. Available from: `http://www.bts.gov/programs/geographic_information_services/download_sites/gdt/maindownload.html`. 101, 115, 240

[CCN+99] M. Carey, D. Chamberlin, S. Narayanan, B. Vance, D. Doole, S. Rielau, R. Swagerman, and N. Mattos. O-O, What Have They Done to DB2? In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases*, pages 542–553, Edinburgh, Scotland, UK, 1999. Morgan Kaufmann Publishers. Available from: `http://www.vldb.org/conf/1999/P51.pdf`. 155

[CDK05] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems – Concepts and Design*. International Computer Science Series. Addison-Wesley, Reading, MA, USA, 4th edition, 2005. 191

[Cel05] J. Celko. *Joe Celko's SQL for Smarties – Advanced SQL Programming*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 3rd edition, 2005. 77

[CGA02] CGAL Consortium. *CGAL Developers' Manual, Version 2.3*, 2002. Available from: `http://www.cgal.org/Manual/doc_pdf/Developers_manual.pdf`. 185

[CGA04] CGAL Consortium. *CGAL User and Reference Manual, Version 3.1*, 2004. Available from: `http://www.cgal.org/Manual/doc_pdf/cgal_manual.pdf`. 140, 151, 153

[CGG+95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-Memory Graph Algorithms. In *SODA'95, Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, San Francisco, CA, USA, 1995. ACM Press. Available from: `ftp://ftp.cs.brown.edu/pub/papers/compgeo/extmem-soda95.ps.Z`. 123

[Che76] P. P. Chen. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976. 29

[CLR01] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 2001. 80, 82, 101, 103, 108, 171

[CMW87] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A Graphical Query Language Supporting Recursion. In *SIGMOD'87, Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data*, pages 323–330, San Francisco, CA, USA, 1987. ACM Press. 84

[Coh81] D. Cohen. On Holy Wars and a Plea for Peace. *IEEE Computer Magazine*, 1981. 60, 143, 201

[Con97] S. Conrad. *Föderierte Datenbanksysteme – Konzepte der Datenintegration*. Springer-Verlag, Berlin, Germany, 1997. (in German). 211

[Dad96] P. Dadam. *Verteilte Datenbanken und Client/Server-Systeme – Grundlagen, Konzepte und Realisierungsformen*. Springer-Verlag, Berlin, Germany, 1996. (in German). 191

[DD00] C. J. Date and H. Darwen. *Foundation for Future Database Systems – The Third Manifesto*. Addison-Wesley, Reading, MA, USA, 2nd edition, 2000. 39

[Deu04] S. Deumlich. Google-Wrapper für DB2. Study thesis, Information Integration Group, University of Berlin, Germany, 2004. Available from: `http://www.informatik.hu-berlin.de/mac/lehre/diplom/studarbeit_google.pdf`. 221

[DG96] L. P. Deutsch and J.-L. Gailly. ZLIB Compressed Data Format Specification Version 3.3. Request for Comments (RFC) 1950, Internet Engineering Task Force, 1996. Available from: `http://www.gzip.org/zlib/rfc1950.pdf`. 171

[Die05] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer Verlag, Heidelberg, Germany, 3rd edition, 2005. Available from: `http://www.math.uni-hamburg.de/home/diestel/books/graph.theory`. 77, 79

[Dij59] E. W. Dijkstra. A Note on two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. 82

[DLW02] L. Daniel, P. Loree, and A. Whitener. *Inside MapInfo Professional*. Thomson Delmar Learning, Clifton Park, NY, USA, 3rd edition, 2002. 4

[EG94] M. Erwig and R. H. Güting. Explicit Graphs in a Functional Model for Spatial Databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):787–804, 1994. Available from: `http://web.engr.oregonstate.edu/~erwig/papers/XGraphsInSpatialDB_REPORT.pdf`. 83

[EH90] M. J. Egenhofer and J. R. Herring. Categorizing Binary Topological Relationships Between Regions, Lines, and Points in Geographic Databases. Technical report, Department of Surveying Engineering, University of Maine, Orono, ME, USA, 1990. Available from: `http://www.spatial.maine.edu/~max/9intReport.pdf`. 27

[EPS04] European Petrol Survey Group. *Using the EPSG Geodetic Parameter Dataset*, 2004. Available from: `http://www.epsg.org/guides/docs/G7-1.pdf`. 38, 51, 202, 205

[ESR98] Environmental Systems Research Institute, Inc., Redlands, CA, USA. *ESRI Shapefile Technical Description*, 1998. Available from: `http://www.esri.com/library/whitepapers/pdfs/shapefile.pdf`. 36, 47, 63, 70, 115, 162, 227, 260

[FFS00] J.-C. Freytag, M. Flasza, and M. Stillger. Implementing Geospatial Operations in an Object-Relational Database System. In *SSDBM 2000, Proceedings of the 12th International Conference on Scientific and Statistical Database Management*, pages 209–219, Berlin, Germany, 2000. IEEE Computer Society. Available from: `http://www.dbis.informatik.hu-berlin.de/fileadmin/research/papers/conferences/2000-ssdbm-freytag.pdf`. 46, 169

[FMS98] E. Feuerstein and A. Marchetti-Spaccamela. Memory Paging for Connectivity and Path Problems in Graphs. *Journal of Graph Algorithms and Applications*, 2(2), 1998. Available from: `http://www.cs.brown.edu/publications/jgaa/accepted/98/FeuersteinMarchetti.2.6.pdf`. 123

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible Hashing – A Fast Access Method for Dynamic Files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979. 103

[GB91] O. Günther and J. Bilmes. Tree-Based Access Methods for Spatial Databases – Implementation and Performance Evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):342–356, 1991. 124

[GBE+00] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, 25(1):1–42, 2000. Available from: `http://www.informatik.fernuni-hagen.de/import/pi4/papers/TODS.pdf`. 23

[GBL95] M. Graves, E. R. Bergeman, and C. B. Lawrence. A Graph-Theoretic Data Model for Genome Mapping Databases. In *HICSS'95, Proceedings of the 28th Annual Hawaii International Conference on System Sciences*, volume 5, pages 32–41, Kihei, HI, USA, 1995. IEEE Computer Society. Available from: `http://csdl.computer.org/comp/proceedings/hicss/1995/6921/00/69210032.pdf`. 77

[Geo05] GeoAPI Working Group of the Open Geospatial Consortium Technical Committee. *GeoAPI, Version 2.0*, 2005. Available from: `http://geoapi.sourceforge.net/2.0/javadoc/index.html`. 56

[Gep02] A. Geppert. *Objektrelationale und objektorientierte Datenbankkonzepte und -systeme*. dpunkt.verlag, Heidelberg, Germany, 2002. (in German). 154

[GHH+03] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel. Boolean Operations on 3D Selective Nef Complexes – Data Structure, Algorithms, and Implementation. In *ESA'03, Proceedings of the 11th Annual European Symposium Algorithms*, volume 2832 of *Lecture Notes in Computer Science*, pages 654–666, Budapest, Hungary, 2003. Springer Verlag. Available from: `http://www.mpi-sb.mpg.de/~kettner/pub/nef_3d_ecgtr_03.pdf`. 153

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissidis. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995. 21, 64, 69, 105, 133, 162

[GJSB05] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, Boston, MA, USA, 3rd edition, 2005. Available from: `http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf`. 17, 43, 55, 102

[Gol91] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, 1991. Available from: `http://www.dacya.ucm.es/dani/goldbergsun.pdf`. 60, 261

[Gol06a] S. Goldau. Reimplementierung der SQL-Typhierarchie für räumliche Daten. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2006. in preparation (in German). 39

[Gol06b] C. Gollmick. *Konzept, Realisierung und Anwendung nutzerdefinierter Replikation in mobilen Datenbanksystemen*. PhD thesis, Database and Information Systems Group, University of Jena, Germany, 2006. (in German). Available from: `http://www.db-thueringen.de/servlets/DerivateServlet/Derivate-8296/gollmick-dissertation.pdf`. 254

[GPBG94] M. Gyssens, J. Paredaens, J. V. van Bussche, and D. V. Gucht. A Graph-Oriented Object Database Model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, 1994. 84

[GRA05] GRASS GIS 6.1 reference manual [online]. 2005. Available from: `http://grass.itc.it/grass61/manuals/html61_user/index.html`. 4, 7, 222, 249, 285

[GRA06] GRASS 6 programmer's manual – GIS library [online]. 2006. Available from: `http://mpa.itc.it/markus/grass60progman/`. 225, 239

[GSVGM98] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity Search in Databases. In *VLDB'98, Proceedings of 24th International Conference on Very Large Data Bases*, pages 26–37, New York, NY, USA, 1998. Morgan Kaufmann Publishers. Available from: `http://www.vldb.org/conf/1998/p026.pdf`. 84, 124

[Gün93] O. Günther. Efficient Computation of Spatial Joins. In *ICDE 1993, Proceedings of the 9th International Conference on Data Engineering*, pages 50–59, Vienna, Austria, 1993. IEEE Computer Society. 26

[Gut84] A. Guttman. R-Trees – A Dynamic Index Structure for Spatial Searching. In *SIGMOD 1984, Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, USA, 1984. ACM Press. 16, 124, 239

[Güt94a] R. H. Güting. An Introduction to Spatial Database Systems. *VLDB Journal*, 3(4):357–399, 1994. Available from: `http://www.vldb.org/journal/VLDBJ3/P357.pdf`. 4

[Güt94b] R. H. Güting. GraphDB – Modeling and Querying Graphs in Databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, pages 297–308, Santiago de Chile, Chile, 1994. Morgan Kaufmann Publishers. Available from: `http://www.vldb.org/conf/1994/P297.PDF`. 83

[Haj70] G. Hajos. *Einführung in die Geometrie*. Teubner Verlagsgesellschaft, Leipzig, Germany, 1970. (in German). 131

[Has97] D. A. Hastings. The Geographic Information Systems – GRASS HOWTO [online]. 1997. Available from: `http://www.tldp.org/HOWTO/GIS-GRASS/index.html`. 223

[Hen05] T. Henke. Erweiterungsmöglichkeiten von Data-Mining-Systemen für das räumliche datenumfeld. Master's thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 286

[Het05] M. Hetterle. Heterogene Replikation räumlicher Daten zwischen Informix IDS und DB2 UDB. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 267, 268

[HFK06] J. A. Hernandez, Martinez F., and J. Keogh. *SAP R/3 Handbook*. McGraw-Hill Osborne Media, Emeryville, CA, USA, 3rd edition, 2006. 213

[HJR97] Y.-W. Huang, N. Jing, and E. A. Rundensteiner. Integrated Query Processing Strategies for Spatial Path Queries. In *ICDE'97, Proceedings of the 13th International Conference on Data Engineering*, pages 477–486, Birmingham, UK, 1997. IEEE Computer Society. Available from: `http://davis.wpi.edu/dsrg/UMICH/ssdbm97.ps`. 83

[HK04] J. Han and M. Kamber. *Data Mining – Concepts and Techniques*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004. 286

[HK05] P. Hachenberger and L. Kettner. Boolean Operations on 3D Selective Nef Complexes – Optimized Implementation and Experiments. In *SPM'05, Proceedings of the 9th ACM symposium on Solid and Physical Modeling*, pages 163–174, Cambridge, MA, USA, 2005. ACM Press. Available from: `http://www.mpi-sb.mpg.de/~kettner/pub/nef_3d_experiments_spm_05.pdf`. 171

[HL93] J. Hoschek and D. Lasser. *Fundamentals of Computer Aided Design*. AK Peters, Ltd., Wellesley, MA, USA, 1993. 128, 151

[Hud05] P. Hudson. *PHP in a Nutshell*. O'Reilly Media, Sebastopol, CA, USA, 2005. Available from: `http://hudzilla.org/phpbook/`. 76

[iAn04] iAnywhere Solutions, Inc., Dublin, CA, USA. *iAnywhere MobiLink Synchronization User's Guide, Version 9*, 2004. Available from: `http://www.ianywhere.com/developer/product_manuals/sqlanywhere/0901/en/pdf/dbmlen9.pdf`. 257

[IBM04a] International Business Machines Corp., Armonk, NY, USA. *IBM DB2 Information Integrator – Federated Systems Guide, Version 8.2*, 2004. Available from: `ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/iiyfpe81.pdf`. 192, 213

[IBM04b] International Business Machines Corp., Armonk, NY, USA. *IBM DB2 Information Integrator – SQL Replication Guide and Reference, Version 8.2*, 2004. Available from: `ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/db2e0e82.pdf`. 251, 257, 262, 274

[IBM04c] International Business Machines Corp., Armonk, NY, USA. *IBM DB2 Information Integrator – Wrapper Development Guide, Version 8.2*, 2004. Available from: `ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/iiywde80.pdf`. 218, 249

[IBM04d] International Business Machines Corp., Armonk, NY, USA. *IBM DB2 Spatial Extender and Geodetic Extender – User's Guide and Reference, Version 8.2*, 2004. Available from: `ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/db2sbe81.pdf`. 6, 36, 38, 39, 46, 100, 114, 150, 169, 178, 194, 204, 229, 246, 284

[IBM04e] International Business Machines Corp., Armonk, NY, USA. *IBM DB2 Universal Database – SQL Reference Volume 1, Version 8.2*, 2004. see also [IBM04f]. Available from: `ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/db2s1e81.pdf`. 37, 124, 310

[IBM04f] International Business Machines Corp., Armonk, NY, USA. *IBM DB2 Universal Database – SQL Reference Volume 2, Version 8.2*, 2004. see also [IBM04e]. Available from: `ftp://ftp.software.ibm.com/ps/products/db2/info/vr82/pdf/en_US/db2s2e81.pdf`. 310

[IBM04g] International Business Machines Corp., Armonk, NY, USA. *IBM DB2 Universal Database Net Search Extender – Administration and User's*

Guide, Version 8.2, 5th edition, 2004. Available from: `http://publibfp.boulder.ibm.com/epubs/pdf/h1267404.pdf`. 123

[IBM05] International Business Machines Corp. *WebSphere MQ – Application Programming Guide, Version 6.0*, 2005. Available from: `http://publibfp.boulder.ibm.com/epubs/pdf/csqzal10.pdf`. 254

[IEE85] IEEE. *Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Std 754:1985, New York, NY, USA, 1985. 63, 261

[IFX02a] International Business Machines Corp., Armonk, NY, USA. *IBM Informix Geodetic DataBlade Module, Version 3.11*, 2002. Available from: `http://publibfp.boulder.ibm.com/epubs/pdf/8675.pdf`. 48

[IFX02b] International Business Machines Corp., Armonk, NY, USA. *IBM Informix Spatial DataBlade Module, Version 8.20*, 2002. Available from: `http://publibfp.boulder.ibm.com/epubs/pdf/9119.pdf`. 6, 48, 56, 59, 114, 127, 192, 197, 204

[IFX04a] International Business Machines Corp., Armonk, NY, USA. *IBM Informix Guide to SQL – Reference, Version 10.0*, 2004. Available from: `http://publib.boulder.ibm.com/infocenter/idshelp/v10/topic/com.ibm.pdfs.doc/25122830.pdf`. 197

[IFX04b] International Business Machines Corp., Armonk, NY, USA. *IBM Informix R-Tree Index User's Guide, Version 10.0*, 2004. Available from: `http://publib.boulder.ibm.com/infocenter/idshelp/v10/topic/com.ibm.pdfs.doc/25122970.pdf`. 246

[IFX05] International Business Machines Corp., Armonk, NY, USA. *IBM Informix Dynamic Server Enterprise Replication Guide, Version 10.0*, 2nd edition, 2005. Available from: `http://publib.boulder.ibm.com/infocenter/idshelp/v10/topic/com.ibm.pdfs.doc/25122791.pdf`. 7, 258, 275

[Ign01] T. Ignatova. Object-relational image extensions in 2d electrophoresis gels databases. Master's thesis, Dababase and Information Systems Group, University of Rostock, Germany, 2001. 15, 287

[Int02] International Business Machines Corp. San Francisco maps city services with IBM DB2 and DB2 Spatial Extender [online]. 2002. Available from: `http://www.esri.com/partners/hardware/ibm/pdfs/sf.pdf`. 191

[ISO98] ISO/IEC 13249-4 WD. *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 4: General Purpose Facilities*, 1998. discontinued. 23

[ISO99] ISO/IEC 9075-2:1999. *Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation)*, 1st edition, 1999. 16

[ISO01] ISO/IEC 9075-2:2001 WD. *Information Technology – Database Languages – SQL – Part 7: Temporal (SQL/Temporal)*, 2001. discontinued. 23

[ISO02] ISO/IEC 13249-6:2002. *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 6: Data Mining*, 1st edition, 2002. 286

[ISO03a] ISO/DIS 19107:2003. *Geographic Information – Spatial Schema*, 1st edition, 2003. 23

[ISO03b] ISO/DIS 19111:2003. *Geographic Information – Spatial Referencing by Coordinates*, 1st edition, 2003. 23, 38

[ISO03c] ISO/IEC 13249-2:2003. *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 2: Full-Text*, 2nd edition, 2003. 125

[ISO03d] ISO/IEC 13249-3:2003. *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 3: Spatial*, 2nd edition, 2003. 4, 15, 19, 26, 59, 60, 75, 77, 125, 128, 129, 132, 143, 150, 186, 192, 200, 281

[ISO03e] ISO/IEC 13249-5:2003. *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 5: Still Image*, 2nd edition, 2003. 287

[ISO03f] ISO/IEC 9075-10:2003. *Information Technology – Database Languages – SQL – Part 10: Object Language Bindings (SQL/OLB)*, 2nd edition, 2003. 56, 75, 282

[ISO03g] ISO/IEC 9075-11:2003. *Information Technology – Database Languages – SQL – Part 11: Information and Definition Schemas (SQL/Schemata)*, 1st edition, 2003. 29, 31

[ISO03h] ISO/IEC 9075-13:2003. *Information Technology – Database Languages – SQL – Part 13: SQL Routines and Types Using the Java Programming Language (SQL/JRT)*, 2nd edition, 2003. 57

[ISO03i] ISO/IEC 9075-2:2003. *Information Technology – Database Languages – SQL – Part 2: Foundation (SQL/Foundation)*, 2nd edition, 2003. 9, 16, 19, 27, 36, 43, 58, 133, 137, 146, 168, 198, 226, 265

[ISO03j] ISO/IEC 9075-3:2003. *Information Technology – Database Languages – SQL – Part 3: Call-Level Interface (SQL/CLI)*, 3rd edition, 2003. 55

[ISO03k] ISO/IEC 9075-4:2003. *Information Technology – Database Languages – SQL – Part 4: Persistent Stored Modules (SQL/PSM)*, 3rd edition, 2003. 102

[ISO03l] ISO/IEC 9075-9:2003. *Information Technology – Database Languages – SQL – Part 9: Management of External Data (SQL/MED)*, 2nd edition, 2003. 18, 211, 213, 249, 284

[ISO04a] ISO 19125-1:2004. *Geographic Information – Simple Feature Access – Part 1: Common Archtecture*, 1st edition, 2004. 20, 64, 317

[ISO04b] ISO 19125-2:2004. *Geographic Information – Simple Feature Access – Part 2: SQL Option*, 1st edition, 2004. 317

[ISO05a] ISO/IEC 13249-3:2005 WD. *Information Technology – Database Languages – SQL Multimedia and Application Packages – Part 3: Spatial*, 3rd edition, 2005. working draft. 80, 84, 85, 128, 140, 143, 146

[ISO05b] ISO/TC 211/WG 4/PT 19136 DIS. *Geographic Information – Geography Markup Language (GML)*, 1st edition, 2005. 44, 59, 61, 132, 260

[JHR96] N. Jing, Y.-W. Huang, and E. A. Rundensteiner. Hierarchical Optimization of Optimal Path Finding for Transportation Applications. In *CIKM'96, Proceedings of the 5th International Conference on Information and Knowledge Management*, pages 261–268, Rockville, MD, USA, 1996. ACM Press. Available from: `http://widit.slis.indiana.edu/irpub/CIKM/1996/pdf30.pdf`. 124

[JSHL02] V. Josifovski, P. M. Schwarz, L. M. Haas, and E. T. Lin. Garlic – A New Flavor of Federated Query Processing for DB2. In *SIGMOD 2002, Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 524–532, Madison, WI, USA, 2002. ACM Press. Available from: `http://portal.acm.org/ft_gateway.cfm?id=564751&type=pdf&CFID=73987646`. 220

[JTS03a] Vivid Solutions. *JTS Topology Suite – Developer's Guide, Version 1.4*, 2003. Available from: `http://www.vividsolutions.com/jts/bin/JTS%20Developer%20Guide.pdf`. 69

[JTS03b] Vivid Solutions. *JTS Topology Suite – Technical Specifications, Version 1.4*, 2003. Available from: `http://www.vividsolutions.com/jts/bin/JTS%20Technical%20Specs.pdf`. 56, 63

[Kac03] H. Kache. Interoperable geoprocessing between spatial databases. Master's thesis, Database and Information Systems Group, Brandenburg University of Technology, Cottbus, Germany, 2003. 221, 228

[Kar89] M. S. Karasick. *On the Representation and Manipulation of Rigid Solids.* PhD thesis, Departement of Computer Science, McGill University, Montreal, QC, Canada, 1989. 142, 146

[KBL05] M. Kifer, A. Bernstein, and P. M. Lewis. *Database Systems – An Application-Oriented Approach.* Addison-Wesley, Boston, MA, USA, 2nd edition, 2005. 55

[Ket99] L. Kettner. Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces. *Computational Geometry – Theory and Applications*, 13(1):65–90, 1999. Available from: `http://www.mpi-sb.mpg.de/~kettner/pub/polyhedron_cgta_99.ps.gz`. 152

[Kim05] W. Kim. On Metadata Management Technology – Status and Issues. *Journal of Object Technology*, 4(2):41–48, 2005. Available from: `http://www.jot.fm/issues/issue_2005_03/column4/column4.pdf`. 203

[KK00] M. Kennedy and S. Kopp. *Understanding Map Projections.* ESRI press, 2000. 13

[Kle97] R. Klein. *Algorithmische Geometrie.* Addison-Wesley, Bonn, Germany, 1997. 98

[KPS00] H.-P. Kriegel, M. Pötke, and T. Seidl. Managing Intervals Efficiently in Object-Relational Databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*, pages 407–418, Cairo, Egypt, 2000. Morgan Kaufmann Publishers. Available from: `http://www.vldb.org/conf/2000/P407.pdf`. 169

[KR88] B. W. Kernighan and D. M. Ritchie. *Programming in C.* Prentice Hall, Englewood Cliffs, NJ, USA, 1988. 17, 55, 159

[KST01] P. Z. Kunszt, A. S. Szalay, and A. R. Thakar. The Hierarchical Triangular Mesh. In *Mining the Sky – Proceedings of the MPA/ESO/MPE workshop*, pages 631–637, Garching, Germany, 2001. Springer-Verlag. Available from: `http://www.mpa-garching.mpg.de/~cosmo/kunszt.ps.gz`. 97

[Lar78] Per-Åke Larson. Dynamic Hashing. *BIT*, 18(2):184–201, 1978. 103

[Leh02] W. Lehner. *Subskriptionssysteme - Marktplatz für omnipräsente Informationen.* Teubner Verlagsgesellschaft, Stuttgart, Germany, 2002. (in German). 253

[Lig02] P. Liggesmeyer. *Software-Qualität – Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag, Heidelberg, Germany, 2002. (in German). 192

[LSC94] D.-R. Liu, S. Shekhar, and M. Coyle. An Evaluation of Access Methods for Spatial Networks. In *ICDE'94, Proceedings of the 10th International Conference on Data Engineering*, Houston, TX, USA, 1994. IEEE Computer Society. Available from: `ftp.cs.umn.edu/dept/users/shekhar/ccam.acmGis94.ps.Z`. 124

[Luf05] J. Lufter. *Unterstützung komplexer Datenstrukturen in SQL-Norm und objektrelationalen Datenbanksystemen*. PhD thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 16, 19

[LWS01] C. Lange, A. Winter, and H. M. Sneed. Comparing Graph-Based Program Comprehension Tools to Relational Database-Based Tools. In *IWPC 2001, Proceedings of the 9th International Workshop on Program Comprehension*, pages 209–220, Toronto, ON, Canada, 2001. IEEE Computer Society. Available from: `http://www.uni-koblenz.de/~clange/IWPCPaper.pdf`. 77

[Mac99] A. MacDonald, editor. *Building a Geodatabase – ArcInfo 8*. ESRI press, Redlands, CA, USA, 1999. 4

[Man99] G. Manzini. An Analysis of the Burrows-Wheeler Transform. In *SODA'99, Proceedings of the 10th ACM-SIAM Symposium on Discrete Algorithms*, pages 669–677, Baltimore, MD, USA, 1999. SIAM Press. Available from: `http://www.mfn.unipmn.it/~manzini/papers/bwjacm2.pdf`. 171

[Map01] MapInfo Corp., Troy, NY, USA. *SpatialWare 4.5 Server for IBM DB2 – Release Notes*, 2001. Available from: `http://testdrive.mapinfo.com/TECHSUPP/MIPROD.NSF/0/a7b8977f1f91cff485256a020055fe04/%24FILE/release45_d.pdf`. 49

[Map04] MapInfo Corp., Troy, NY, USA. *MapInfo SpatialWare for Microsoft SQL Server – v4.8 User Guide*, 2004. Available from: `http://www.mapinfo.com/documentation/software/spatialware_for_sql_server/english/4_8/sw48_userguide.pdf`. 43, 49

[ML75] W. D. Maurer and T. G. Lewis. Hash Table Methods. *ACM Computing Surveys*, 7(1):5–19, 1975. 103

[MN99] K. Mehlhorn and S. Näher. *LEDA – A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, UK, 1999. 151

[Mol98] M. Molenaar. *An Introduction to the Theory of Spatial Object Modelling for GIS*. Taylor & Francis Ltd., London, UK, 1998. 4

[MS90] M. V. Mannino and L. D. Shapiro. Extensions to Query Languages for Graph Traversal Problems. *IEEE Transactions on Knowledge and Data Engineering*, 2(3), 1990. 84

[MyS05] MySQL AB, Uppsala, Sweden. *MySQL 5.0 Reference Manual*, 2005. Available from: `http://downloads.mysql.com/docs/refman-5.0-en.a4.pdf`. 50, 202, 211, 222

[Nef78] W. Nef. *Beiträge zur Theorie der Polyeder*. Herbert Lang, Bern, Switzerland, 1978. (in German). 152, 153

[NGV93] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for External Graph Searching. In *PODS'93, Proceedings of the 12th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 222–232, Washington, DC, USA, 1993. ACM Press. Available from: `http://www.cs.duke.edu/~jsv/Papers/NGV93.graphsearching.pdf`. 123

[NH02] H. Niemann and W. Hasselbring. Adaptive Replikationsstrategie für heterogene, autonome Informationssysteme. In *Tagungsband zum 14. GI-Workshop Grundlagen von Datenbanken*, Ostseebad Dierhagen-Strand, Germany, 2002. Fachbereich Informatik, Universität Rostock. (in German). Available from: `http://wwwdb.informatik.uni-rostock.de/wsgvdb02/papers/Niemann.ps`. 254

[Nie05] S. Niebius. Nutzungspotentiale räumlicher Daten in einem Data Warehouse. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 286

[Nil80] N. J. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, Berlin, Germany, 1980. 82

[NM04] M. Neteler and H. Mitasova. *Open Source GIS – A GRASS GIS Approach*. SECS, The Kluwer international series in Engineering and Computer Science. Kluwer Academic Publishers/Springer, Boston, MA, USA, 2nd edition, 2004. 223

[OGC99] Open Geospatial Consortium. *OpenGIS Simple Features Specification for SQL, Revision 1.1*, 1999. Available from: `http://portal.opengeospatial.org/files/?artifact_id=829`. 5, 16, 20, 25, 44, 48, 50, 52, 69, 132, 193, 200, 206

[OGC04] Open Geospatial Consortium. *OpenGIS Geography Markup Language (GML) Encoding Specification – Implementation Specification, Revision 3.1.1*, 2004. Available from: `http://portal.opengeospatial.org/files/?artifact_id=4700`. 44, 48, 61, 132, 186

[OGC05a] Open Geospatial Consortium. *OpenGIS Implementation Specification for Geographic Information – Simple Feature Access – Part 1: Common Architecture, Version 1.1.0*, 2005. see also [ISO04a]. Available from: `http://portal.opengeospatial.org/files/?artifact_id=13227`. 56

[OGC05b] Open Geospatial Consortium. *OpenGIS Implementation Specification for Geographic Information – Simple Feature Access – Part 2: SQL option, Version 1.1.0*, 2005. see also [ISO04b]. Available from: `http://portal.opengeospatial.org/files/?artifact_id=13228`. 57

[Ora03] Oracle Corp., Redwood City, CA, USA. *Oracle Spatial – Best Practices*, 2003. Available from: `http://www.oracle.com/technology/products/spatial/pdf/spatial_best_practices.pdf`. 204

[Ora05a] Oracle Corp., Redwood City, CA, USA. *Empowering Applications with Spatial Analysis and Mining*, 2005. Available from: `http://otn.oracle.com/products/spatial/pdf/10g_spatial_analysis_mining_twp.pdf`. 286

[Ora05b] Oracle Corp., Redwood City, CA, USA. *Oracle Database – Advanced Replication, 10g Release 2 (10.2)*, 2005. Available from: `http://download.oracle.com/docs/cd/B19306_01/server.102/b14226.pdf`. 7, 251, 259

[Ora05c] Oracle Corp., Redwood City, CA, USA. *Oracle Database – SQL Reference, 10g Release 2 (10.2)*, 2005. Available from: `http://download.oracle.com/docs/cd/B19306_01/server.102/b14200.pdf`. 51

[Ora05d] Oracle Corp., Redwood City, CA, USA. *Oracle Spatial – GeoRaster, 10g Release 2 (10.2)*, 2005. Available from: `http://download.oracle.com/docs/pdf/B14254_01.pdf`. 52, 287

[Ora05e] Oracle Corp., Redwood City, CA, USA. *Oracle Spatial – Topology and Network Data Models, 10g Release 2 (10.2)*, 2005. Available from: `http://download.oracle.com/docs/pdf/B14256_01.pdf`. 6, 52, 84, 85, 90

[Ora05f] Oracle Corp., Redwood City, CA, USA. *Oracle Spatial – User's Guide and Reference, 10g Release 2 (10.2)*, 2005. Available from: `http://download.oracle.com/docs/pdf/B14255_01.pdf`. 51, 114, 127, 193, 198, 204, 221

[Ore86] J. A. Orenstein. Spatial Query Processing in an Object-Oriented Database System. In *SIGMOD 1986, Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 326–336, Washington, D.C., USA, 1986. ACM Press. Available from: `http://www.cs.cmu.edu/~christos/courses/826-resources/PAPERS+BOOK/z-order.PDF`. 26

[OSQZ02] P. van Oosterom, J. Stoter, W. Quak, and S. Zlatanova. The Balance Between Geometry and Topology. In *SDH 2002, Proceedings of the 10th International Symposium on Spatial Data Handling*, pages 209–224, Ottawa, ON, Canada, 2002. Springer-Verlag. Available from: `http://www.isprs.org/commission4/proceedings/pdfpapers/147.pdf`. 85, 152, 154

[Ott06] S. Ott. Performanceevaluation verschiedener Indexmechanismen für räumliche Daten basierend auf DB2 UDB. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2006. (in German). 169, 195

[Pag06] M. Pagel. Implementierung einer Erweiterung von DB2 UDB zur Verwaltung von Rasterdaten. Master's thesis, Dababase and Information Systems Group, University of Jena, Germany, 2006. (in German). 287

[Pöt01] M. Pötke. *Spatial Indexing for Object-Relational Databases*. PhD thesis, Database Systems Group 2, University of Munich, Germany, 2001. 127, 128

[PS93] P. P. Preparata and M. I. Shamos. *Computational Geometry – An Introduction*. Springer-Verlag, New York, NY, USA, 1993. 11

[PSQ05] The PostgreSQL Global Development Group. *PostgreSQL 8.1.0 Documentation*, 2005. Available from: `http://www.postgresql.org/files/documentation/pdf/8.1/postgresql-8.1-A4.pdf`. 52, 200

[PZMT03] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query Processing in Spatial Network Databases. In *VLDB 2003, Proceedings of 29th International Conference on Very Large Data Bases*, Berlin, Germany, 2003. Available from: `http://www.vldb.org/conf/2003/papers/S24P02.pdf`. 77, 101

[Ref05] Refractions Research, Inc., Victoria, BC, Canada. *PostGIS Manual*, 2005. Available from: `http://postgis.refractions.net/docs/postgis.pdf`. 52, 187, 198, 200, 225

[SA02] K. Stolze and D. W. Adler. *Reducing Index Size for Multi-Level Grid Indexes*. International Business Machines Corp., 2002. U.S. Patent Application 20030212650, Serial No. 10/141919. 47

[Saf05a] Safe Software, Inc., Surrey, BC, Canada. *Feature Manipulation Engine (FME) – Readers and Writers*, 2005. Available from: `ftp://ftp.safe.com/fme/2006/docs/FMEReadersWriters.pdf`. 260

[Saf05b] Safe Software, Inc., Surrey, BC, Canada. *FME Fundamentals*, 2005. Available from: `ftp://ftp.safe.com/fme/2006/docs/FME_Fundamentals.pdf`. 191, 251

[Sag94] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, NY, USA, 1994. 124, 169

[Sal05] C. Salomon. Replikation räumlicher Daten vor Oracle nach DB2 UDB. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 270, 273, 274

[Sal06a] C. Salomon. Modellierung räumlicher Daten in DB2 UDB für den performanten Zugriff einer Web-Anwendung. Master's thesis, Database and Information Systems Group, University of Jena, Germany, 2006. (in German). 40

[Sal06b] M. Salzbrenner. Spezifikation und Implementierung einer JAVA-Schnittstelle für die Repräsentation von räumlichen Daten in einer Anwendung. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2006. (in German). 63, 66, 69, 71

[Sam84] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, 1984. 16

[Sár01] Ferenc Sárközy. Some remarks on the development of the spatial data model [online]. 2001. Available from: `http://www.agt.bme.hu/public_e/mod/datamodel.htm`. 4

[SB06] K. Stolze and R. Bittner. Integrating 3D Spatial Operations in Relational Database Systems. *Special Issue of Computational Geometry – Theory and Applications on CGAL – the Computational Geometry Algorithms Library*, 2006. submitted. 151, 156

[SC02] K. Stolze and R. A. Coss. *User-defined Aggregate Functions in Database Systems without Native Support*. International Business Machines Corp., 2002. U.S. Patent Application 20030233380, Serial No. 10/173402. 91

[Sch06] I. Schmitz. Optimierung von Datenbankanwendungen aus der Astronomie am Beispiel des ROSAT-Datenbankkatalogs. Master's thesis, Database Systems Group 3, University of Munich, Germany, 2006. (in German). 96, 97, 150

[Sed88] R. Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, USA, 2nd edition, 1988. 80, 82

[Sel02] D. Selman. *Java 3D Programming – A guide to key concepts and effective techniques*. Manning Publications, Greenwich, CT, USA, 2002. 183

[Sew05] J. Seward. *bzip2 and libbzip2, version 1.0.3 – A program and library for data compression*, 2005. Available from: `http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.pdf`. 171

[SHN04] T. Schwarz, M. Hönle, N. Großmann, and D. Nicklas. A Library for Managing Spatial Context Using Arbitrary Coordinate Systems. In *PerCom 2004, Proceedings of the 2nd IEEE Conference on Pervasive Computing and Communications Workshops*, pages 48–54, Orlando, FL, USA, 2004. IEEE Computer Society. Available from: `ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/INPROC-2004-19/INPROC-2004-19.pdf`. 202

[SK01] K. Stolze and K. Kulkarni. *SQL/MM Spatial Change Proposal – Introducing spatial information schema*. ISO, Victoria, BC, Canada, 2001. Document yyj057. 48

[SL97] S. Shekhar and D.-R. Liu. CCAM – A Connectivity-Clustered Access Method for Networks and Network Computations. *IEEE Transactions on Knowledge and Data Engineering*, 9(1):102–119, 1997. Available from: `ftp://ftp.cs.umn.edu/dept/users/shekhar/ccam.TKDE96.ps`. 123

[Sno95] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. SECS, The Kluwer international series in Engineering and Computer Science. Kluwer Academic Publishers, Boston, MA, USA, 1995. 23

[SRC02a] K. Stolze, F. Y. Rao, and Y. Chen. *Systems, Methods, and Computer Program Products to Improve Indexing of Multidimensional Databases*. International Business Machines Corp., 2002. U.S. Patent No. 6,941,319. 47

[SRC02b] K. Stolze, F. Y. Rao, and Y. Chen. *Systems, Methods and Computer Program Products to Reduce Computer Processing in Grid Cell Size Determination for Indexing of Multidimensional Databases*. International Business Machines Corp., 2002. U.S. Patent Application 20030212677, Serial No. 10/144389. 47

[SS04] T. Steinbach and K. Stolze. *DB2 Index Extensions by Example and in Detail*. IBM DeveloperWorks, 2004. Available from: `http://www.ibm.com/developerworks/db2/library/techarticle/dm-0312stolze/`. 169

[Sta05] StarQuest Ventures, Inc., Inverness, CA, USA. *SQDR Plus for iSeries*, 2005. Available from: `http://www.starquest.com/Supportdocs/techsqdr400/SQDRPlus_iSeries.pdf`. 259, 262, 267, 274, 275

[Sto01] K. Stolze. SQL/MM Part 5: Still Image – The Standard and Implementation Aspects. In *BTW 2001, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 9. BTW-Konferenz*, Informatik Aktuell, pages 345–363, Oldenburg, Germany, 2001. Springer-Verlag. 287

[Sto02] K. Stolze. Still Image Extensions in Database Systems – A Product Overview. *Datenbank-Spektrum*, pages 40–47, 2002. Available from: `http://www.datenbank-spektrum.de/pdf/dbs-02-40.pdf`. 287

[Sto03a] K. Stolze. Integrating custom geocoders with the DB2 Spatial Extender [online]. 2003. Available from: `http://www.ibm.com/developerworks/db2/library/techarticle/0305stolze/0305stolze.html`. 48

[Sto03b] K. Stolze. SQL/MM Spatial – The Standard to Manage Spatial Data in Relational Database Systems. In *BTW 2003, Datenbanksysteme für Business, Technologie und Web, Tagungsband der 10. BTW-Konferenz*, volume 26 of *Lecture Notes in Informatics*, pages 247–264, Leipzig, Germany, 2003. Springer-Verlag. Available from: `http://doesen0.informatik.uni-leipzig.de/proceedings/paper/68.pdf`. 5, 39

[Sto04] K. Stolze. Replicating spatial data in DB2 UDB [online]. 2004. Available from: `http://www.ibm.com/developerworks/db2/library/techarticle/dm-0402stolze/`. 258, 262, 264, 276

[Sto05a] K. Stolze. DB2 Spatial Extender performance tuning [online]. 2005. Available from: `http://www.ibm.com/developerworks/db2/library/techarticle/dm-0510stolze/`. 47

[Sto05b] K. Stolze. Extending the DB2 Spatial Extender – add your customized spatial functionality to the DB2 Spatial Extender [online]. 2005. Available from: `http://www.ibm.com/developerworks/db2/library/techarticle/dm-0511stolze/`. 58, 95, 113, 160

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, USA, 3rd edition, 1997. 55, 152, 159

[Sun01] Sun Microsystems, Inc., Palo Alto, CA, USA. *Java JDBC Data Access API Specification, Version 3.0*, 2001. Available from: `http://javashoplm.sun.com/ECom/docs/Welcome.jsp?StoreId=22&PartDetailId=7076-jdbc-3.0-fr-spec-oth-JSpec&TransactionId=noreg`. 55, 56, 75

[SWCD97] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *ICDE'97, Proceedings of the 13th International Conference on Data Engineering*, pages 422–432, Birmingham, UK, 1997. IEEE Computer Society. Available from: `http://www.cs.uic.edu/~sistla/movingobjects.ps`. 4, 23

[Tel06] Tele Atlas North America. GDT Dynamap [online]. 2006. Available from: `http://www.teleatlas.com/Pub/Products/Map_Databases/Dynamap/index.htm`. 115

[TJS97] V. J. Tsotras, C. S. Jensen, and R. T. Snodgrass. A Notation for Spatiotemporal Queries. Technical report, TimeCenter, 1997. Available from: `http://www.cs.auc.dk/research/DP/tdb/TimeCenter/TimeCenterPublications/TR-10.pdf`. 23

[Tob70] W. R. Tobler. A Computer Movie Simulating Urban Growth in the Detroit Region. *Economic Geography*, 46-2:234–240, 1970. Available from: `http://www.geog.ucsb.edu/~tobler/publications/pdf_docs/geog_analysis/ComputerMovie.pdf`. 26

[TS06] C. Türker and G. Saake. *Objektrelationale Datenbanken – Ein Lehrbuch*. dpunkt.verlag, Heidelberg, Germany, 2006. (in German). 18

[W3C03] World Wide Web Consortium (W3C). *Scalable Vector Graphics (SVG) 1.1 Specification*, 2003. Available from: `http://www.w3.org/TR/SVG11/REC-SVG11-20030114.pdf`. 36, 63

[Wal05] M. Walther. Ein Kostenmodell zum Tunen von Grid-Indizes des DB2 Spatial Extenders. Master's thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 169, 178

[WCO00] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly Media, Sebastopol, CA, USA, 3rd edition, 2000. Available from: `http://safari.oreilly.com/?XmlId=0-596-00027-8`. 76

[Wes04] R. West. *ArcGIS 9 – Understanding ArcSDE*. ESRI press, 2004. 139

[Wit05] S. Witzel. Verarbeitung von Graphen in relationalen Datenbankmanagementsystemen. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2005. (in German). 102, 103, 123

[WS06] P. Wessel and W. H. F. Smitth. *The Generic Mapping Tools (GMT) – Technical Reference and Cookbook, Version 4.1.1*, 2006. Available from: `http://gmt.soest.hawaii.edu/gmt/doc/pdf/GMT_Docs.pdf`. 12

[Zen06] C. Zentgraf. Abbildung von Geometrien in relationalen Datenbanksystemen auf Graphen. Study thesis, Database and Information Systems Group, University of Jena, Germany, 2006. (in German). 102, 106, 121

[ZK02] J. Zytkow and W. Klösgen, editors. *Handbook of Data Mining and Knowledge Discovery*. Oxford University Press, Oxford, UK, 2002. 286

[ZRS02] S. Zlatanova, A. A. Rahman, and W. Shi. Topology for 3D Spatial Objects. In *ISG 2002, International Symposium and Exhibition on Geoinformation*, Kuala Lumpur, Malaysia, 2002. Available from: `http://www.gdmc.nl/zlatanova/thesis/html/refer/ps/SZ_AR_WS_02.pdf`. 77, 131

[ZZ94] J. L. Zhao and A. Zaki. Spatial Data Traversal in Road Map Databases – A Graph Indexing Approach. In *CIKM'94, Proceedings of the 3rd International Conference on Information and Knowledge Management*, pages 355–362, Gaithersburg, MD, USA, 1994. ACM Press. Available from: `http://portal.acm.org/citation.cfm?id=191308`. 83

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel und Literatur angefertigt habe.

Jena, 10. Juli 2006

Lebenslauf

Persönliche Daten

Name	Knut Stolze
Geburtsdatum	08. Januar 1975
Geburtsort	Jena, Thüringen

Schulische und universitäre Ausbildung

1981 – 1983	Polytechnische Oberschule "Dr. Theodor Neubauer"
1983 – 1993	Polytechnische Oberschule "Julius Schaxel" mit erweitertem Russischunterricht (1991 umbenannt in Staatliches Gymnasium "Ernst Haeckel"); Abschluss 10. Klasse (1991) und Abitur (1993)
1994 – 1999	Diplomstudium Informatik mit Nebenfach Mathematische Biologie an der Friedrich-Schiller-Universität Jena Vertiefungsrichtung: Datenbanken und Informationssysteme, Abschluss als Diplom-Informatiker (Dipl.-Inf.)

Beruflicher Werdegang

1993 – 1994	Wehrdienst, absolviert bei der Panzerpionierkompanie 380 in Weißenfels
1993 – 1998	Mitarbeiter und Assistent bei der Volker Alex OHG (McDonald's Jena)
1998 – 1999	Studentische Hilfskraft bei Energie Systeme Nord GmbH
1999 – 2002	Gastwissenschaftler (visting scientist) im IBM Silicon Valley Laboratory in San Jose, CA, USA
seit 2002	Wissenschaftlicher Mitarbeiter am Lehrstuhl für Datenbanken und Informationssysteme der Fakultät für Mathematik und Informatik, Friedrich-Schiller-Universität Jena
seit 2002	Mitarbeiter bei der IBM Deutschland Entwicklung GmbH

Jena, 10. Juli 2006